Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Факультет вычислительной математики и информатики
Кафедра экономико-математических методов и статистики

РАБОТА ПРОВЕРЕНА
Рецензент,

_____

« » _____ 2015 г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой, д.ф.-м.н., профессор
_____ А.В. Панюков

« » _____ 2015 г.

Параллельная реализация метода эллипсоидов для задач оптимизации большой размерности

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ ЮУрГУ-010400.62.2015.001.001 ВКР

Руководитель проекта, к.ф.-м.н., доцент
_____ В.А. Голодов
« » _____ 2015 г.

Автор проекта
студент группы ВМИ-413
_____ В.А. Безбородов
« » _____ 2015 г.

Нормоконтролер, к.ф.-м.н., доцент
_____ Т.А. Макаровских
« » _____ 2015 г.

Челябинск, 2015

Дипломная работа выполнена мной совершенно самостоятельно. Все использованные в работе материалы и концепции из опубликованной научной литературы и других источников имеют ссылки на них.

_____ В.А. Безбородов
« »_____2015 г.

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Факультет вычислительной математики и информатики
Кафедра экономико-математических методов и статистики

УТВЕРЖДАЮ
Заведующий кафедрой, д.ф.-м.н., профессор
_____ А.В. Панюков
« » _____2015 г.

З А Д А Н И Е

на выпускную квалификационную работу студента

Безбородова Вячеслава Александровича

Группа ВМИ-413

1. Тема работы: Параллельная реализация метода эллипсоидов для задач оптимизации большой размерности.
   Утверждена приказом по университету от « » _____2015 г. №_____
2. Срок сдачи студентом законченной работы « » _____2015 г.
3. Исходные данные к работе
   3.1. Модельные данные;
   3.2. Типовые примеры;
   3.3. Самостоятельно сконструированные тестовые данные.
4. Перечень вопросов, подлежащих разработке
   4.1. Изучение общей схемы работы метода эллипсоидов;

4.2. Изучение приемов параллельной обработки данных;

4.3. Разработка класса (типа данных) для реализации параллельно выполняемых операций над матрицами с применением библиотеки GMP;

4.4. Разработка параллельной реализации метода эллипсоидов для задачи выпуклого программирования;

4.5. Тестирование разработанного программного обеспечения;

4.6. Проверка на модельных данных.

5. Перечень графического материала

5.1. Оценка сложности некоторых алгоритмов поиска – 1 л.

5.2. Оценка сложности некоторых алгоритмов сортировки – 1 л.

5.3. Иллюстрация парадигмы Fork-Join – 1 л.

5.4. Размер и диапазон чисел с плавающей запятой в языке C++ – 1 л.

5.5. Способы разбиения элементов матрицы: горизонтальный, вертикальный и блочный – 1 л.

5.6. Время выполнения операции сложения матриц – 1 л.

5.7. Время выполнения операции умножения матриц – 1 л.

5.8. Ускорение операции сложения матриц – 1 л.

5.9. Ускорение операции умножения матриц – 1 л.

5.10. Зависимость времени выполнения операции умножения матриц размерности 500×500 от числа используемых потоков – 1 л.

5.11. Графическое представление задачи оптимизации – 1 л.

5.12. Время выполнения оптимизации – 1 л.

6. Календарный план

Наименование этапов дипломной работы	Срок выполнения этапов работы	Отметка о выполнении
1. Сбор материалов и литературы по теме дипломной работы	02.02.2015 г.	
2. Исследование способов построения математической модели задачи	10.02.2015 г.	
3. Разработка математической модели и алгоритма	22.02.2015 г.	
4. Реализация разработанных алгоритмов	04.03.2015 г.	
5. Проведение вычислительного эксперимента	15.03.2015 г.	
6. Подготовка пояснительной записки дипломной работы	21.03.2015 г.	
Написание главы 1	03.04.2015 г.	
Написание главы 2	25.04.2015 г.	
Написание главы 3	13.05.2015 г.	
7. Оформление пояснительной записки	18.05.2015 г.	
8. Получение отзыва руководителя	23.05.2015 г.	
9. Проверка работы руководителем, исправление замечаний	28.05.2015 г.	
10. Подготовка графического материала и доклада	01.06.2015 г.	
11. Нормоконтроль	05.06.2015 г.	
12. Рецензирование, представление зав. кафедрой	10.06.2015 г.	

7. Дата выдачи задания « »_____2015 г.

Заведующий кафедрой_____/А.В. Панюков /

Руководитель работы_____/В.А. Голодов /

Студент_____/В.А. Безбородов /

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Факультет вычислительной математики и информатики
Кафедра экономико-математических методов и статистики

АННОТАЦИЯ

Безбородов, В.А. Параллельная реализация метода эллипсоидов для задач оптимизации большой размерности / В.А. Безбородов. – Челябинск: ЮУрГУ, Факультет вычислительной математики и информатики, 2015. – 80 с., 9 ил., 3 табл., 3 прил., библиогр. список – 35 названий.

В дипломной работе произведен анализ вычислительной сложности выполняемых операций алгоритма метода эллипсоидов. На основе результатов анализа разработана параллельная реализация метода эллипсоидов, адаптированная для решения задач оптимизации большой размерности на многопроцессорных и/или многоядерных вычислительных системах с общей разделяемой памятью.

Приведены результаты вычислительных экспериментов, продемонстрирован пример решения задачи оптимизации большой размерности с применением разработанной программной реализации.

ОГЛАВЛЕНИЕ

Введение

Задачи оптимизации получили чрезвычайно широкое распространение в технике, экономике, управлении. Типичными областями применения теории оптимизации являются прогнозирование, планирование промышленного производства, управление материальными ресурсами, а также контроль качества выпускаемой продукции [21].

Успешность хозяйственной деятельности зависит от того, как распределяются имеющиеся ограниченные ресурсы. В связи с тем, что такая задача оптимального распределения довольно часто возникает на практике в различных сферах жизнедеятельности, актуальным становится поиск способов ускорения ее решения.

Задачи оптимизации большой размерности характеризуются высокой трудоемкостью. Использование доступного ресурса аппаратного параллелизма современных вычислительных систем рассматривается как возможность ускорения поиска их решения. Применение библиотек, реализующих поддержку арифметики произвольной точности, диктуется необходимостью достижения высокой точности при решении практических задач.

Одним из известных методов оптимизации является метод эллипсоидов, важной характеристикой которого является доказанная полиномиальная сходимость. Разрабатывали, а впоследствии развивали метод эллипсоидов такие ученые, как Шор Н.З. [33], Юдин Д.Б., Немировский А.С. [35], Хачиян Л.Г. [28], Гершович В.И. [19], Стецюк П.И. [27] и др.

В работе исследован алгоритм метода эллипсоидов. Разработана его программная реализация, ориентированная на многопроцессорные и/или многоядерные вычислительные системы с общей разделяемой памятью. Показано приложение программной реализации к решению задачи оптимизации большой размерности.

Целями работы являются:

1) разработка параллельной реализации метода эллипсоидов, поддерживающей арифметику произвольной точности;

2) использование полученной реализации метода эллипсоидов для решения задачи оптимизации большой размерности.

В соответствии с поставленными целями в работе решаются следующие **задачи**:

- исследование операций классического алгоритма метода эллипсоидов на вычислительную сложность;
- разработка программной реализации алгоритма с распараллеливанием наиболее длительных по времени операций;
- обеспечение поддержки арифметики расширенной и произвольной точности;
- демонстрация использования разработанного ПО для решения задачи оптимизации большой размерности;
- проверка и тестирование разработанного программного обеспечения.

Объектом исследования данной работы является метод эллипсоидов, **предметом** – параллельная реализация метода, поддерживающая арифметику произвольной точности.

Работа состоит из введения, 3 глав, заключения, 3 приложений и списка литературы. Объем работы составляет 80 страниц. Список литературы содержит 35 наименования.

В первой главе рассматривается алгоритм метода эллипсоидов, производится анализ вычислительной сложности входящих в него операций с целью поиска наиболее ресурсоемких, нуждающихся в ускорении путем распараллеливания.

Во второй главе описывается способ параллельной реализации метода эллипсоидов, аргументированно доказывается необходимость обеспечения поддержки арифметики неограниченной точности, а также описывается тестовое окружение разрабатываемого класса.

В третьей главе приводится параллельная реализация алгоритма метода эллипсоидов, детально рассматривается пример решения модельной оптимизационной задачи, демонстрируется решение задачи оптимизации большой размерности.

В заключении перечислены основные результаты работы.

1 Метод эллипсоидов

В настоящее время большое внимание уделяется созданию автоматизированных систем планирования, проектирования и управления в различных областях промышленности. На первый план выдвигаются вопросы качества принимаемых решений, в связи с чем возрастает роль методов и алгоритмов решения оптимизационных задач в математическом обеспечении автоматизированных систем различного уровня и назначения.

Имеется несколько основных источников, порождающих задачи оптимизации: задачи математического программирования, задачи нелинейного программирования, задачи оптимального управления, задачи дискретного программирования или задачи смешанного дискретно-непрерывного типа [34].

Сфера применения методов оптимизации огромна. Создание эффективных методов оптимизации является ключом к решению многих вычислительных проблем математического программирования, особенно для задач большой размерности.

Для минимизации гладких функций широко применяются различные модификации градиентных процессов, поскольку направление антиградиента в данной точке локально является направлением наискорейшего спуска. Регулировка шага в большинстве алгоритмов этого типа основана на том, чтобы обеспечить монотонное и в достаточной степени «существенное» уменьшение значения функции на каждом шаге.

Простейший обобщенный градиентный метод состоит в движении на каждом шаге в направлении, обратном направлению обобщенного градиента. Этот метод под названием обобщенного градиентного спуска (ОГС) предложен Н.З. Шором в 1961 г. в связи с необходимостью разработки эффективного алгоритма решения транспортных задач большой размерности для задач текущего планирования, решаемых в Институте кибернетики АН УССР совместно с Госпланом УССР. Впервые метод обобщенного градиентного спуска для минимизации кусочно-линейных выпуклых функций использовался при решении транспортных и транспортно-производственных задач [29]. Затем метод ОГС был распространен на класс произвольных выпуклых функ-

ций [30] и на задачи выпуклого программирования в гильбертовом пространстве [25]. Широкое распространение получили стохастические аналоги ОГС [22]. Алгоритмы решения задач математического программирования, построенные на основе ОГС, отличаются простотой и, что особенно важно для задач большой размерности, экономным использованием оперативной памяти ЭВМ.

К ограничениям метода ОГС относятся его довольно медленная сходимость, сложность контроля точности решения и то, что он применим только к классу выпуклых функций.

В 1969–1970 гг. Н.З. Шором были предложены ускоренные варианты обобщенных градиентных методов, основанные на использовании операции растяжения пространства в направлении градиента и разнсти двух последовательных градиентов [31]. Идея этих методов существенно отлична от той, которая используется для ускорения сходимости в случае гладких функций – идеи квадратичной аппроксимации функции в окрестности минимума, в той или иной мере определяющей формализм как методов сопряженных градиентов, так и квазиньютоновских методов [8]. В то же время предельные варианты методов с растяжением пространства при определенных условиях регулярности и гладкости обладают свойством квадратичной скорости сходимости. Таким образом, предложенные алгоритмы обладают высокой эффективностью и применительно к гладким задачам минимизации. В дальнейшем алгоритмы с растяжением пространства были обобщены на задачи нахождения локальных минимумов невыпуклых негладких функций [32].

В США и Западной Европе градиентными методами минимизации негладких функций всерьез начали заниматься примерно с 1973 г. сначала в связи с приложениями в области дискретного программирования [6], а затем в целом для решения задач большой размерности [7]. Результаты работ в этом направлении на Западе достаточно полно представлены в сборнике [11]. Особенно интенсивно развивается направление так называемой ε-субградиентной оптимизации, по идее близкое, с одной стороны, к алгоритмам В.Ф. Демьянова решения минимаксных задач, а с другой, особенно в формальном отноше-

нии, – к алгоритмам метода сопряженных градиентов (или «давидоновского» типа).

И наконец, в последнее время обнаружились очень интересные связи между алгоритмами последовательных отсечений и алгоритмами с растяжением пространства [33, 35].

Таким образом, область обобщенных градиентных методов оптимизации не представляет нечто окончательно сформировавшееся и застывшее, а, наоборот, быстро развивается.

1.1 Алгоритм метода эллипсоидов

Рассмотрим алгоритм решения задачи выпуклого программирования, гарантирующий уменьшение объема области, в которой локализуется оптимум, со скоростью геометрической прогрессии, причем знаменатель этой прогрессии зависит только от размерности задачи. Этот алгоритм относится к классу алгоритмов обобщенного градиентного спуска с растяжением пространства в направлении градиента (ОГСРП) [33].

Пусть имеется задача выпуклого программирования:

$$\min f_0(x) \tag{1.1}$$

при ограничениях

$$f_i(x) \leq 0, \quad i = 1, \ldots, m, \quad x \in E_n, \tag{1.2}$$

где E_n – евклидово пространство размерности n, $f_\nu(x)$, $\nu = \overline{0, m}$ – выпуклые функции, определенные на E_n; $g_\nu(x)$ – субградиенты соответствующих функций. Пусть имеется априорная информация о том, что существует оптимальная точка $x^* \in E_n$ (не обязательно единственная), которая находится в шаре радиуса R с центром в точке x_0 (формально к системе ограничений 1.2 можно добавить ограничение $||x - x_0|| \leq R$).

Рассмотрим следующий итеративный алгоритм (при $n > 1$).

Алгоритм 1 Метод эллипсоидов

Шаг 0. Инициализация.
Положить $x_k = x_0$; $B_k = E$, где E – единичная матрица размерности $n \times n$; $h_k = \frac{R}{n+1}$ – коэффициент, отвечающий за уменьшение объема шара. Перейти к шагу 1.
Шаг 1. Вычислить

$$g(x_k) = \begin{cases} g_0(x_k), & \text{если } \max\limits_{1 \le i \le m} f_i(x_k) \le 0, \\ g_{i^*}(x_k), & \text{если } \max\limits_{1 \le i \le m} f_i(x_k) = f_{i^*}(x_k) > 0. \end{cases}$$

Если $g(x_k) = 0$, то завершить алгоритм; x_k – оптимальная точка. Иначе перейти к шагу 2.
Шаг 2. Вычислить $\xi_k = \dfrac{B_k^T g(x_k)}{||B_k^T g(x_k)||}$. Перейти к шагу 3.
Шаг 3. Вычислить $x_{k+1} = x_k - h_k \cdot B_k \cdot \xi_k$. Перейти к шагу 4.
Шаг 4. Вычислить $B_{k+1} = B_k \cdot R_\beta(\xi_k)$, где $R_\beta(\xi_k)$ – оператор растяжения пространства в направлении ξ_k с коэффициентом β (см. определение 1.1), $\beta = \sqrt{\frac{n-1}{n+1}}$. Перейти к шагу 5.
Шаг 5. Вычислить $h_{k+1} = h_k \cdot r$, где $r = \frac{n}{\sqrt{n^2-1}}$. Перейти к шагу 1.

Определение 1.1. Оператором растяжения пространства E_n в направлении ξ с коэффициентом β называется оператор $R_\beta(\xi)$, действующий на вектор x следующим образом [34]:

$$R_\beta(\xi)x = \left(E + (\beta - 1)\xi_k \xi_k^T\right) x.$$

Рассмотрим вопрос оценки скорости сходимости метода эллипсоидов. Покажем, что данный вариант алгоритма ОГСРП сходится по функционалу со скоростью геометрической прогрессии, причем знаменатель этой прогрессии зависит только от размерности задачи.

Лемма 1.1. Последовательность $\{x_k\}_{k=0}^{\infty}$, генерируемая алгоритмом 1, удовлетворяет неравенству

$$||A_k(x_k - x^*)|| \le h_k \cdot (n+1), \quad A_k = B_k^{-1}, \quad k = 0, 1, 2, \ldots \tag{1.3}$$

Доказательство леммы для краткости изложения опущено и может быть найдено в [33].

Множество точек x, удовлетворяющих неравенству

$$||A_k(x_k - x)|| \leq (n+1)h_k = R \cdot \left(\frac{n}{\sqrt{n^2-1}}\right)^k,$$

представляет собой эллипсоид Φ_k, объем которого $v(\Phi_k)$ равен

$$\frac{v_0 R^n \left(\frac{n}{\sqrt{n^2-1}}\right)^{nk}}{\det A_k},$$

где v_0 – объем единичного n-мерного шара. Получаем

$$\frac{v(\Phi_{k+1})}{(\Phi_k)} = \frac{\left(\frac{n}{\sqrt{n^2-1}}\right)^n \cdot \det A_k}{\det A_{k+1}} = \frac{\left(\frac{n}{\sqrt{n^2-1}}\right)^n \cdot \det A_k}{\det R_\alpha(\xi_k) \cdot \det A_k} = \frac{1}{\alpha}\left(\frac{n}{\sqrt{n^2-1}}\right)^n =$$

$$= \sqrt{\frac{n-1}{n+1}}\left(\frac{n}{\sqrt{n^2-1}}\right)^n = q_n < 1.$$

Таким образом, объем эллипсоида, в котором локализуется оптимальная точка x^* в соответствии с неравенством (1.3), убывает со скоростью геометрической прогрессии со знаменателем q_n.

1.2 Вычислительная сложность операций метода эллипсоидов

В индустрии разработки программного обеспечения известен феномен, который состоит в том, что на 20% методов программы приходится 80% времени ее выполнения [2]. Такое эмпирическое правило хорошо согласуется с более общим принципом, известным как принцип Парето либо «правило 80/20».

Утверждение 1 (Принцип Парето). 80% результата можно получить, приложив 20% усилий.

Относящийся не только к программированию, этот принцип очень точно характеризует оптимизацию программ [24]. В работе [10] Дональд Кнут

указал, что менее 4% кода обычно соответствуют более чем 50% времени выполнения программы. Опираясь на принцип Парето, можно сформулировать последовательность действий, приводящих к ускорению работы имеющихся программ: необходимо найти в коде «горячие точки» и сосредоточиться на оптимизации наиболее трудоемких процессов.

Проанализируем подробнее вычислительную сложность операций алгоритма метода эллипсоидов. Для этого введем некоторые обозначения из области асимптотического анализа [20].

Определение 1.2. Функция $f(n)$ *ограничена сверху* функцией $g(n)$ асимптотически с точностью до постоянного множителя, т.е.

$$f(n) = O\left(g(n)\right),$$

если существуют целые N и K, такие, что $|f(n)| \leq Kg(n)$ при всех $n \geq N$.

Определение 1.3. Функция $f(n)$ *ограничена снизу* функцией $g(n)$ асимптотически с точностью до постоянного множителя, т.е.

$$f(n) = \Omega\left(g(n)\right),$$

если существуют целые N и K, такие, что $f(n) \geq Kg(n)$ при всех $n \geq N$.

Определение 1.4. Функция $f(n)$ *ограничена снизу и сверху* функцией $g(n)$ асимптотически с точностью до постоянного множителя, т.е.

$$f(n) = \Theta\left(g(n)\right),$$

если одновременно выполнены условия определений 1.2 и 1.3.

На **шаге 1** алгоритма 1 изменение текущего субградиента $g(x_k)$ происходит на основании анализа значений функций ограничений (1.2) в текущей точке x_k, т.е. анализируется последовательность значений

$$\max_{1 \leq i \leq m} f_i(x_k).$$

Вычислительная сложность поиска максимального из m чисел зависит от выбора алгоритма.

При использовании линейного последовательного поиска цикл выполнит m итераций. Трудоемкость каждой итерации не зависит от количества элементов, поэтому имеет сложность $T^{iter} = O(1)$. В связи с этим, верхняя оценка всего алгоритма поиска $T_m^{min} = O(m) \cdot O(1) = O(m \cdot 1) = O(m)$. Аналогично вычисляется нижняя оценка сложности, а в силу того, что она совпадает с верхней, можно утверждать $T_m^{min} = \Theta(m)$.

При использовании алгоритма бинарного поиска на каждом шаге количество рассматриваемых элементов сокращается в 2 раза. Количество элементов, среди которых может находиться искомый, на k-ом шаге определяется формулой $\frac{m}{2^k}$. В худшем случае поиск будет продолжаться, пока в массиве не останется один элемент, т.е. алгоритм имеет логарифмическую сложность: $T_m^{binSearch} = O\left(\log(m)\right)$. Резюмируем все вышесказанное относительно алгоритмов поиска в виде таблицы 1.

Таблица 1 — Оценка сложности некоторых алгоритмов поиска

Алгоритм	**Структура данных**	**Временная сложность**		**Сложность по памяти**
		В среднем	**В худшем**	**В худшем**
Линейный поиск	Массив из n элементов	$O(n)$	$O(n)$	$O(1)$
Бинарный поиск	Отсортированный массив из n элементов	$O\left(\log(n)\right)$	$O\left(\log(n)\right)$	$O(1)$

Однако при использовании некоторых алгоритмов (например, алгоритма бинарного поиска) потребуются дополнительные процедуры для упорядочивания входной последовательности значений. В таблице 2 представлены асимптотические оценки наиболее известных алгоритмов сортировки массива из n элементов[1].

[1]Более подробный анализ приведен в [18].

Таблица 2 — Оценка сложности некоторых алгоритмов сортировки[1]

Алгоритм	**Временная сложность**			**Сложность по памяти**
	В лучшем	**В среднем**	**В худшем**	**В худшем**
Быстрая сортировка	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$	$O(n)$
Сортировка слиянием	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(n)$
Пирамидальная сортировка	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(1)$
Пузырьковая сортировка	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка вставками	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка выборомм	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Блочная сортировка	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(nk)$
Поразрядная сортировка	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

На **шаге 2** алгоритма производится вычисление нормированного обобщенного градиента

$$\xi_k = \frac{B_k^T g(x_k)}{||B_k^T g(x_k)||},$$

что подразумевает выполнение трудоемких матричных операций, таких как транспонирование, умножение на вектор и на число. Асимптотическая сложность таких операций для матрицы размерности $n \times n$ может быть оценена как $O(n^2)$.

На **шаге 3** при обновлении значения текущей точки выполняется умножение матрицы B_k на вектор ξ_k:

$$x_{k+1} = x_k - h_k \cdot B_k \cdot \xi_k.$$

Аналогично предыдущей оценке, цена такой операции составит $O(n^2)$.

Шаг 4 наиболее сложен из-за необходимости вычисления оператора растяжения пространства

$$B_{k+1} = B_k \cdot R_\beta(\xi_k).$$

Если представить оператор в матричной форме (определение 1.1), то можно видеть, что в процессе его вычисления используются все вышеперечисленные операции, а также операция сложения матриц, сложность которой составляет $O(n^2)$.

На **шаге 5** осуществляется пересчет коэффициента h_k, отвечающего за уменьшение объема шара

$$h_{k+1} = h_k \cdot r.$$

Эта операция может быть выполнена за константное время, т.е. асимптотически ее сложность составит $O(1)$.

Из проведенного анализа вычислительной сложности операций, входящих в алгоритм метода эллипсоидов, можно сделать несколько выводов. Во-первых, учитывая специфику рассматриваемого класса задач (задачи оптимизации большой размерности), наиболее трудоемкие операции будут выполняться значительно дольше менее трудоемких, что приведет к сильно неравномерной загрузке вычислительной системы. Во-вторых, схема чередования сложных/простых в вычислительном смысле операций, а также выбор целевой платформы (многопроцессорные и/или многоядерные системы с общей разделяемой памятью) наталкивают на возможность использования Fork-Join Model (FJM) модели распараллеливания задач для ускорения работы алгоритма метода эллипсоидов.

На основании полученных данных можно сформулировать гипотезу о том, насколько удастся ускорить выполнение метода в целом для решения задач оптимизации большой размерности, если к наиболее ресурсоемким операциям применить алгоритм распараллеливания по данным.

Гипотеза 1 (О соотношении времени). Пусть $f(t)$ – это время работы алгоритма метода эллипсоидов для задачи оптимизации размерности $N \times M$, выполняемого *в однопоточном режиме*. Тогда для параллельной реализации метода эллипсоидов, выполняемой *в многопоточном режиме*, для достаточно больших N и M справедливо равенство

$$F(t) = kf(t),$$

где $F(t)$ – общее время работы параллельной реализации метода, а k – коэффициент ускорения $(k > 1)$.

Приближенная оценка для коэффициента ускорения k может быть получена в ходе выполнения вычислительных экспериментов.

Выводы по главе один

Показано, что широта и разнообразие областей применения методов оптимизации обуславливают необходимость создания эффективных методов оптимизации для решения различных вычислительных проблем математического программирования.

Приведено доказательство того, что метод эллипсоидов гарантированно локализует оптимум со скоростью геометрической прогрессии при условии существования оптимальной точки.

Проведенный анализ метода эллипсоидов показал, что некоторые операции метода вычислительно сложны.

Выдвинута гипотеза о том, что на современных многопроцессорных и/или многоядерных системах с общей разделяемой памятью можно добиться ускорения работы метода эллипсоидов, если распараллелить наиболее ресурсоемкие операции.

2 Ускорение выполнения матричных операций

В процессе решения любой задачи оптимизации возникает необходимость оперирования над матрицами, которые составляются исходя из условий конкретной задачи. Поскольку в данной работе изначально заложена ориентация на решение задач оптимизации большой размерности, то и матрицы, возникающие из анализа условий этих задач, будут иметь большую размерность. Для ускорения работы с такими матрицами необходимы дополнительные меры по их дроблению и одновременной обработке каждой из частей.

Сегодня существует множество способов для организации параллельного исполнения программного кода. Одним из них является модель Fork-Join.

2.1 Парадигма Fork-Join

В параллельном программировании, Fork-Join model (модель ветвление-объединение, FJM) – это способ запуска и выполнения параллельных участков кода, при котором выполнение ветвей завершается в специально обозначенном месте для того, чтобы в следующей точке продолжить последовательное выполнение. Параллельные участки могут разветвляться рекурсивно до тех пор, пока не будет достигнута заданная степень гранулярности задачи. Модель была впервые сформулирована в 1963 г., и может рассматриваться как один из параллельных паттернов проектирования [12].

Различные реализации FJM обычно управляют *задачами*, *волокнами* или *легковесными нитями*, а не *процессами* уровня операционной системы, и используют *пул потоков* для их выполнения. Специальные ключевые слова позволяют программисту определять точки *возможного параллелизма*; проблема создания реальных потоков и управления ими ложится на реализацию.

В [4] приведена иллюстрация парадигмы Fork-Join. Представим ее на рисунке 1. Здесь три участка программы потенциально разрешают параллельное исполнение различных блоков. Последовательное выполнение показано сверху, в то время как его Fork-Join эквивалент снизу.

Легковесные нити, используемые в Fork-Join программировании, обычно имеют свой собственный планировщик, который управляет ими, применяя

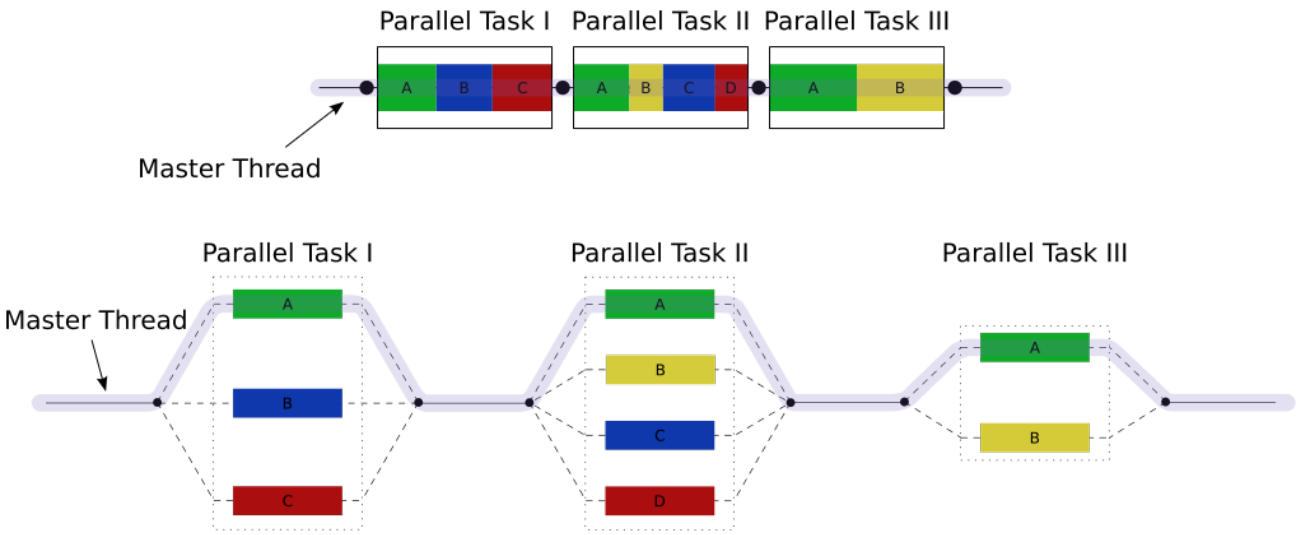

Рисунок 1 — Иллюстрация парадигмы Fork-Join

схему пула потоков. Такой планировщик может быть гораздо проще полнофункционального планировщика задач, применяемого операционной системой. Планировщики потоков общего пользования обязаны приостанавливать и запускать потоки в обозначенных программистом местах, в то время как в парадигме Fork-Join потоки блокируются только в местах непосредственного слияния.

Модель Fork-Join – основная модель параллельного исполнения в технологии OpenMP. Также эта модель поддерживается в Java concurrency framework, в Task Parallel Library для .NET и в Intel Threading Building Blocks. Языки программирования Cilk и Cilk Plus имеют встроенную поддержку FJM в форме ключевых слов *spawn–sync* и *cilk_spawn–cilk_sync* соответственно.

2.2 Необходимость использования библиотеки GMP

Любое численное решение должно достигать определенной точности результата. А при оперировании матрицами большой размерности обеспечение требуемой точности становится еще более актуальным.

Один из способов обеспечения поддержки арифметики расширенной точности заключается в использовании готовых специализированных математических библиотек, таких, например, как GMP.

Библиотека GMP [5] – *бесплатная (свободная)* библиотека для арифметики произвольной точности, выполняемой над знаковыми целыми, рациональными числами и числами с плавающей запятой. При этом практически не существует предела для точности вычислений, если не считать объем доступной памяти ЭВМ, на которой производятся вычисления. GMP имеет богатый набор функций, которые имеют стандартизированный интерфейс.

В основном GMP применяется в криптографических, научно-исследовательских приложениях, приложениях, отвечающих за безопасность в сети Интернет, различных системах вычислительной алгебры и т.д.

Основными целевыми платформами GMP являются Unix-подобные системы, такие как GNU/Linux, Solaris, HP-UX, Mac OS X/Darwin, BSD, AIX и проч. Также поддерживается работа на Windows в 32 и 64-битном режимах.

Библиотека GMP тщательно спроектирована для того, чтобы вычисления производились настолько быстро, насколько это возможно одновременно и для больших, и для малых операндов. Такая скорость возможна благодаря использованию машинных слов в качестве базового арифметического типа и быстрых алгоритмов, включающих высокооптимизированный ассемблерный код для большинства внутренних циклов для целого набора наиболее популярных современных центральных процессоров.

Первая версия GMP вышла в 1991 году. С тех пор библиотека непрерывно улучшается и поддерживается, обеспечивая выход новых версий примерно раз в год.

Начиная с версии 6, GMP распространяется одновременно под двумя лицензиями: GNU LGPL v3 и GNU GPL v2. Такое лицензирование позволяет использовать библиотеку бесплатно, изменять ее и публиковать результат.

Приведем численный пример, наглядно доказывающий необходимость использования библиотеки GMP для поддержки арифметики расширенной и произвольной точности. Рассмотрим следующий код, представленный в листинге 1.

```
1 void MatrixTest::crash_double_type()
2 {
3   std::size_t rows = 2;
4   std::size_t columns = 2;
```

```
5
6     Matrix< double > A( rows, columns );
7     Matrix< double > B( rows, columns );
8
9     MatrixRandomFiller matrix_random_filler;
10    matrix_random_filler.fill( A );
11    matrix_random_filler.fill( B );
12
13    MatrixPrinter matrix_printer( &std::cout );
14    matrix_printer.set_precision( 20 );
15    matrix_printer.print( A );
16    matrix_printer.print( B );
17
18    Matrix< double > C( rows, columns );
19    MatrixMultiplier matrix_multiplier;
20    matrix_multiplier.multiply( C, A, B );
21
22    Matrix< double > D( rows, columns );
23    D = A * B;
24
25    matrix_printer.print( C );
26    matrix_printer.print( D );
27 }
```

Листинг 1 — Исходный код примера

В листинге используются объекты следующих классов:

1) `Matrix< Type >` представляет в программе матрицу, элементы которой имеют тип `Type`;
2) `MatrixRandomFiller` предназначен для заполнения матрицы случайными значениями, равномерно распределенными на интервале $[0, 1)$ (в процессе заполнения используются стандартный генератор псевдослучайных чисел `std::default_random_engine` и равномерное распределение `std::uniform_real_distribution` $P(i|a, b) = \frac{1}{b-a}$ с параметрами $a = 0$, $b = 1$);
3) `MatrixMultiplier` производит перемножение матриц;
4) `MatrixPrinter` отвечает за печать матриц.

В строках 3–4 задается размерность матриц, участвующих в данном примере. Затем происходит создание (строки 6–7), заполнение случайными зна-

чениями (строки 10–11) и печать (строки 15–16) двух операндов. Заметим, что `matrix_printer` настраивается таким образом, чтобы осуществлять вывод в стандартный поток вывода `std::cout` и печатать матрицы с точностью до 20 знаков после запятой. В строке 20 осуществляется умножение матриц `A` и `B` стандартным базовым алгоритмом (строка на столбец), результат которого сохраняется в объекте `C`. В строке 23 также осуществляется умножение операндов. Отличие от строки 20 состоит в том, что в данном случае для умножения используется функция-член класса `Matrix`. Результат второго умножения сохраняется в переменной `D`. После этого осуществляется печать полученных результатов (строки 25–26).

Поскольку матрицы заполняются случайным образом, вывод программы зависит от конкретного запуска. В частности, возможен и такой вариант вывода.

$$\mathtt{A} = \begin{pmatrix} 0.54846871850742517918 & 0.84412510404456542190 \\ 0.95768018538410970564 & 0.80660421324152631328 \end{pmatrix},$$

$$\mathtt{B} = \begin{pmatrix} 0.96126112219013459814 & 0.79203852615307757112 \\ 0.79959221481843290036 & 0.90154689901948148467 \end{pmatrix},$$

$$\mathtt{C} = \begin{pmatrix} 1.20217751736546674124 & 1.19542672538356331557 \\ 1.56553517904925\underline{709736} & 1.48571112974158303643 \end{pmatrix},$$

$$\mathtt{D} = \begin{pmatrix} 1.20217751736546674124 & 1.19542672538356331557 \\ 1.56553517904925\underline{687532} & 1.48571112974158303643 \end{pmatrix}.$$

Из приведенных значений элементов матриц хорошо видно, что для одних и тех же операндов результат перемножения, полученный вне и внутри класса `Matrix`, различен, начиная с 15-го знака после запятой.

Популярный математический сервис WolframAlpha [16] для данных операндов выдает результат

$$\begin{pmatrix} 1.2021775173654667\underline{500} & 1.19542672538356\underline{33424} \\ 1.56553517904925\underline{70011} & 1.48571112974158\underline{29851} \end{pmatrix},$$

в котором наблюдаются отличия в знаках во всех элементах матрицы. Попытка провести аналогичные вычисления на языке программирования R приводит к результату, показанному в листинге 2.

```
> options(digits=20)
> lhs = matrix( c( 0.54846871850742517918, 0.84412510404456542190,
                   0.95768018538410970564, 0.80660421324152631328 ),
                 nrow=2, ncol=2, byrow=TRUE )
> rhs = matrix( c( 0.96126112219013459814, 0.79203852615307757112,
                   0.79959221481843290036, 0.90154689901948148467 ),
                 nrow=2, ncol=2, byrow=TRUE )
> lhs %*% rhs
                      [,1]                 [,2]
[1,] 1.2021775173654667412 1.1954267253835633156
[2,] 1.5655351790492570974 1.4857111297415830364
```

Листинг 2 — Пример вычисления на языке R

Такое расхождение неслучайно. В численном эксперименте (см. листинг 1) участвуют матрицы, все элементы которых имеют тип `double`. В языке C++ для представления чисел с плавающей запятой существуют несколько стандартных типов данных [15]. Характеристики этих типов приведены в таблице 3.

Таблица 3 — Размер и диапазон чисел с плавающей запятой в языке C++

Тип	**Размер, байт**	**Допустимый диапазон значений**
float	4	+/– 3.4e +/– 38 (точность ~7 цифр)
double	8	+/– 1.7e +/– 308 (точность ~15 цифр)
long double	8	+/– 1.7e +/– 308 (точность ~15 цифр)

Согласно информации из таблицы 3, тип `double` имеет точность, в два раза превышающую `float`. В общем случае `double` имеет 15–16 десятичных знаков точности, в то время как `float` только 7.

В численном эксперименте использовался тип данных `double`, что вызвало ошибку округления примерно на 15-м знаке после запятой. Соответственно, если применять тип данных с меньшей (`float`) или большей (`long double`) точностью, меняться будет только положение десятичного знака, на котором будет начинаться расхождение, но *ошибка при этом не исчезнет*.

В такой ситуации разумным видится использование специализированных библиотек, таких как GMP, поддерживающих более точные вычисления. Именно свободный доступ, минимальные ограничения на использование, а также основная функция – поддержка арифметики произвольной точности – делают библиотеку GMP идеальным инструментом для использования в данной работе, особенность которой состоит в оперировании матрицами больших размерностей, что налагает дополнительные (более строгие) ограничения на точность полученного результата.

2.3 Распараллеливание матричных операций

В первой главе показано, что метод эллипсоидов активно использует матричные операции. Вычислительная сложность матричных операций варьируется от $O(n^2)$ (сложение, вычитание, транспонирование) до $O(n^3)$ (умножение). Ускорив выполнение этих операций, можно добиться ускорения выполнение всего метода.

Стандартная библиотека C++ не предоставляет класса для представления матриц в программе. Для языка существуют различные реализации класса матриц, работающие без использования многопоточности [1, 3, 9]. Для ускорения работы метода эллипсоидов требуется разработать класс матриц, поддерживающий распараллеливание матричных операций.

Полный исходный код класса приведен в приложении А. Здесь рассматриваются основные идеи, лежащие в данной реализации.

Класс `Matrix` – шаблонный класс, параметризованный типом `T` элементов матрицы.

```
template< typename T >
class Matrix
```

Класс предоставляет три конструктора. Первый создает матрицу указанной размерности, все элементы которой заполнены начальным значением.

```
Matrix( index_t rows, index_t columns, const value_t& value = value_t( 0 ) )
```

Второй – конструктор копирования – создает матрицу, все элементы которой равны элементам указанной матрицы.

```
Matrix( const my_t& rhs )
```

Третий – конструктор перемещения – берет во владение ресурсы временно созданной матрицы.

```
Matrix( my_t&& rhs )
```

Для индексного доступа к элементам применяются две версии оператора вызова функции (для константных и неконстантных объектов).

```
value_t& operator() ( const index_t& row, const index_t& column )

const value_t& operator() ( const index_t& row, const index_t& column ) const
```

Имеются две версии оператора присваивания (по копии и по временному объекту).

```
my_t& operator= ( const my_t& rhs )

my_t& operator= ( my_t&& rhs )
```

Для класса перегружены основные бинарные операторы.

```
my_t operator+ ( const my_t& rhs ) const

my_t operator- ( const my_t& rhs ) const

my_t operator* ( const my_t& rhs ) const

my_t operator* ( const T& value ) const
```

Операторы сравнения выполняют проверку на равенство элементов двух матриц.

```
bool operator== ( const my_t& rhs ) const

bool operator!= ( const my_t& rhs ) const
```

Для операции транспонирования класс предоставляет функцию.

```
my_t transpose() const
```

Две функции для получения информации о размерности матрицы.

```
std::size_t get_rows() const

std::size_t get_columns() const
```

Это открытый интерфейс класса `Matrix`, доступный пользователю. Помимо перечисленного функционала, класс содержит статическую функцию для доступа к объекту, управляющему механизмом параллельного выполнения операций.

```
static ParallelHandler* get_parallel_handler()
```

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данный момент свидетельствует о наличии *параллелизма по данным* при выполнении матричных расчетов и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между потоками. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*) [14].

При *ленточном* (block–striped) разбиении каждому потоку выделяется то или иное подмножество строк (*rowwise* или *горизонтальное разбиение*) или столбцов (*columnwise* или *вертикальное разбиение*) матрицы (графическая иллюстрация перечисленных подходов представлена на рисунке 2 [23]). Разделение строк и столбцов на полосы в большинстве случаев происходит на *непрерывной* (*последовательной*) основе. При таком подходе для горизонтального разбиения по строкам, матрица A представляется в виде

$$A = (A_0, A_1, \ldots, A_{p-1})^T, A_i = (a_{i_0}, a_{i_1}, \ldots, a_{i_{k-1}}), i_j = ik + j, 0 \leq j < k, k = m/p,$$

где $a_i = (a_{i1}, a_{i2}, \ldots, a_{in}), \quad 0 \leq i < m$, есть i-я строка матрицы A (предполагается, что количество строк m кратно числу вычислительных элементов

(процессоров и/или ядер) p, т.е. $m = k \cdot p$). Существуют и другие способы разбиения, например, блочный.

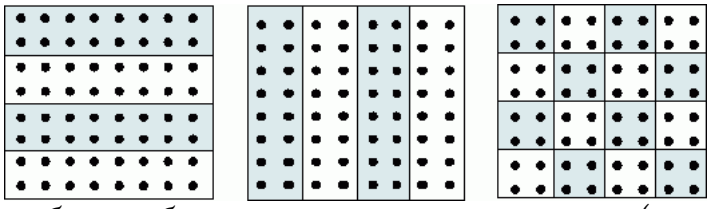

Рисунок 2 — Способы разбиения элементов матрицы (слева направо): горизонтальный, вертикальный и блочный

`ParallelHandler` инкапсулирует параллельное выполнение матричных операций, применяя ленточное разбиение данных на непрерывной основе. Для этого класс предоставляет шаблонную функцию, параметризованную типом итератора и функции.

```
template< typename Iterator, typename Function >
void parallel_for( const Iterator& first,
  const Iterator& last,
  Function&& function )
```

Функция создает и управляет потоками, количество которых зависит от текущих настроек. Класс оперирует понятием *политики распараллеливания*, которое может обозначать один из двух типов: *автоматически* и *вручную*.

```
enum class ParallelPolicy
{
  Automatic,  /**< Calculating based on size of matrix passed to process.  */
  Direct      /**< User defined number of threads.                          */
};
```

Для установки этих и других значений применяется следующий набор функций.

```
void set_auto_parallel_policy();
void set_direct_parallel_policy( std::size_t num_threads );
void set_min_per_thread( std::size_t min_per_thread );
void set_min_num_threads( std::size_t min_num_threads );
```

Функции имеют следующие значения (в порядке объявления): использовать автоматический расчет количества потоков, задать количество потоков вручную, задать минимальную ширину ленты при разбиении данных, установить нижнюю границу для числа используемых потоков.

Для разработанного класса необходимо провести анализ эффективности, доказывающий его преимущество перед последовательным выполнением.

Алгоритмы параллельных матричных операций, основанные на ленточном горизонтальном разбиении матрицы, обладают хорошей «локализацией вычислений», т.е. каждый поток параллельной программы использует только «свои» данные, и ему не требуются данные, которые в данный момент обрабатывает другой поток, нет обмена данными между потоками, не возникает необходимости синхронизации. Это означает, что практически не существуют накладные расходы на организацию параллелизма (за исключением расходов на создание/завершение потоков), и можно ожидать линейного ускорения.

Однако, как видно из представленных ниже результатов, ускорение, которое демонстрируют параллельные матричные операции, далеко от линейного.

Задача сложения матриц обладает сравнительно невысокой вычислительной сложностью – трудоемкость алгоритма имеет порядок $O(n^2)$. Такой же порядок – $O(n^2)$ – имеет и объем данных, обрабатываемый алгоритмом сложения. Время решения задачи одним потоком складывается из времени, когда процессор непосредственно выполняет вычисления, и времени, которое тратится на чтение необходимых для вычислений данных из оперативной памяти в кэш память. При этом время, необходимое для чтения данных, может быть сопоставимо или даже превосходить время счета.

Проведем вычислительный эксперимент: измерим время выполнения последовательного и параллельного алгоритма суммирования и умножения матриц. Столбиковые диаграммы замеров времени выполнения операций сложения и умножения матриц представлены на рисунках 3 и 4 соответственно.

При суммировании элементов матрицы на каждой итерации цикла выполняется простая операция сложения двух чисел. Здесь и далее под *ускорением* выполнения операции будем понимать отношение времени выполнения операции в многопоточном режиме ко времени выполнения той же операции в однопоточном режиме. Достигнутое ускорение для операции сложения матриц различных размерностей представлено на рисунке 5.

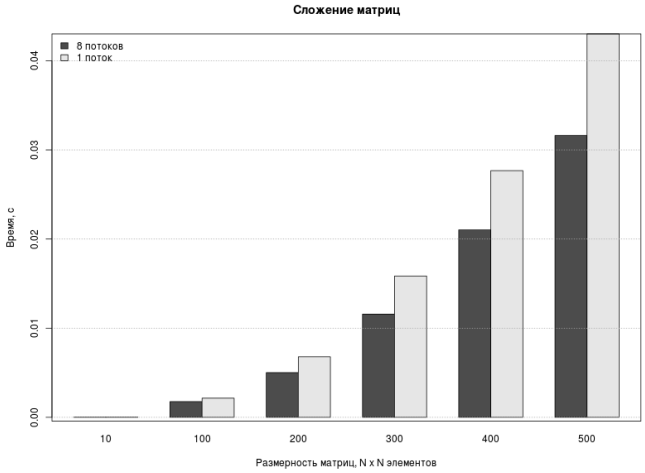

Рисунок 3 — Время выполнения операции сложения матриц

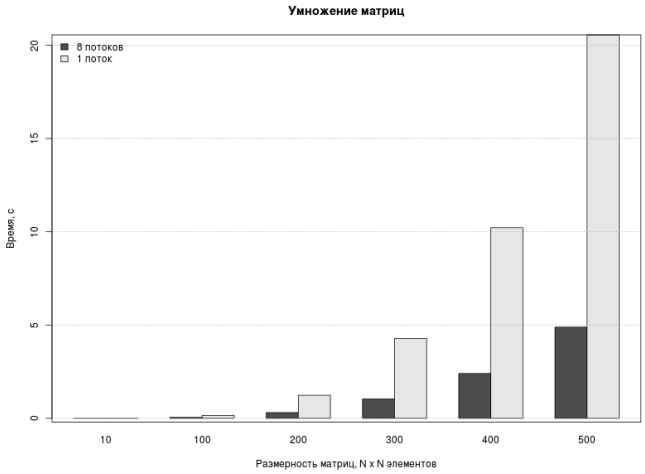

Рисунок 4 — Время выполнения операции умножения матриц

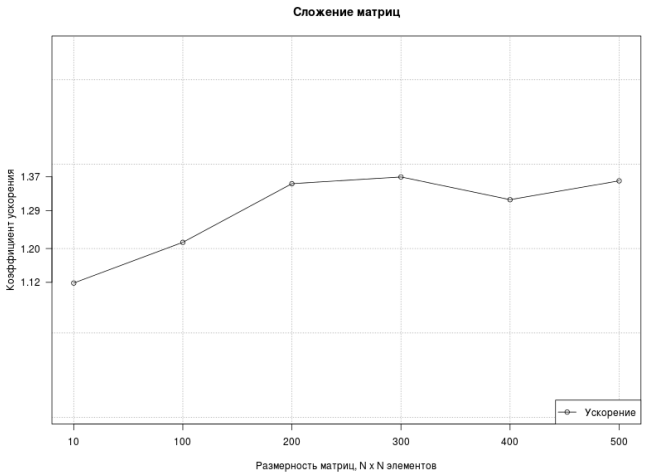

Рисунок 5 — Ускорение операции сложения матриц

Как видно из представленных рисунков, несмотря на меньшую вычислительную сложность, время работы параллельного алгоритма сложения матриц превосходит время выполнения однопоточной версии в среднем всего в 1.3 раза. Этот эксперимент можно рассматривать, как подтверждение предположения о том, что значительная часть времени тратится на выборку необходимых данных из оперативной памяти в кэш процессора.

При умножении матриц на каждой итерации цикла выполняются две операции: более сложная операция умножения и операция сложения. Достигнутое ускорение для операции умножения матриц представлено на рисунке 6.

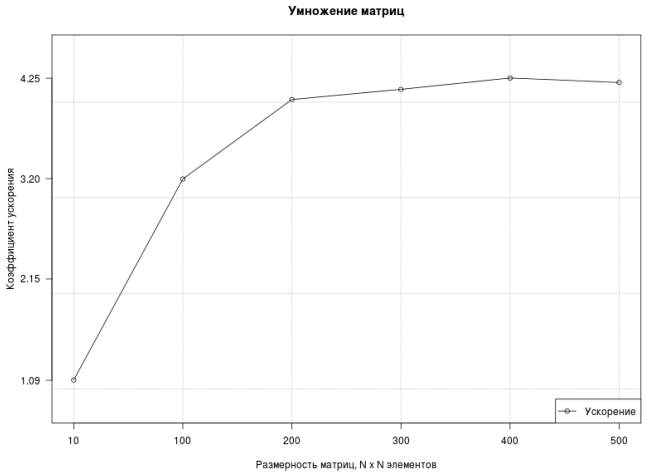

Рисунок 6 — Ускорение операции умножения матриц

Проведем другой эксперимент. Зафиксируем и положим равной 500×500 размерность матриц–операндов. Будем выполнять операцию умножения матриц, применяя каждый раз различное количество потоков, чтобы определить поведение функции времени. Результаты эксперимента представлены на рисунке 7.

Из рисунка 7 видно, что линейное увеличение количества используемых потоков приводит к нелинейному падению времени выполнения операции умножения. Из анализа результатов эксперимента также следует, что использование числа потоков большего, чем аппаратно поддерживается оборудованием, не приведет к дальнейшему падению функции времени. Такое замедление будет возникать из-за частых переключений планировщика потоков для симуляции параллелизма.

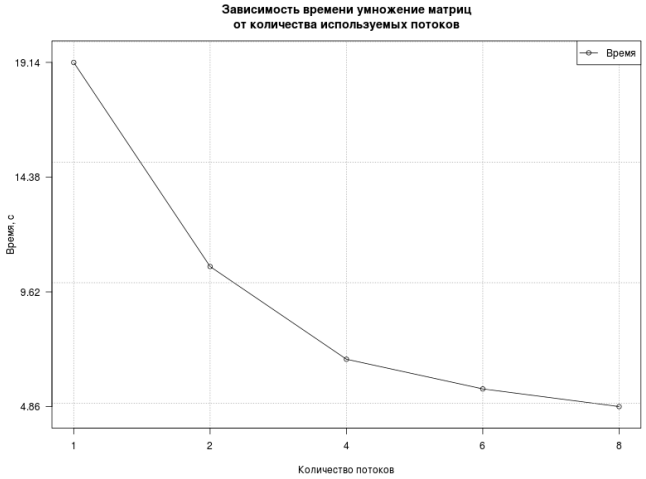

Рисунок 7 — Зависимость времени выполения операции умножения матриц размерности 500×500 от числа используемых потоков

Таким образом, представленная реализация класса, представляющего матрицы в программе, действительно выполняется *быстрее* и имеет *ускорение, большее 1*, благодаря распараллеливанию вычислительно сложных матричных операций.

2.4 Тестирование

Тестирование – важная часть процесса разработки программных продуктов, поскольку ошибки – это не случайное явление.

Различают две формы тестирования [26].

1) *Тестирование методом прозрачного ящика* (white box testing), в котором испытатель имеет понятие о внутренней работе программы.
2) *Тестирование с алгоритмом типа черного ящика* (black box testing), когда программа тестируется без использования информации о ее реализации.

Обе формы тестирования имеют важное значение для проектов, претендующих на попадание в категорию высококачественных. Чаще всего используется тестирование по принципу черного ящика, поскольку оно, как правило, моделирует типичное поведение пользователя. Например, при «черном» тестировании компоненты интерфейса можно рассматривать как кнопки. Если

испытатель, щелкнув по кнопке, не увидел никакой реакции на свое действие, он делает вывод о том, что, очевидно, в программе есть ошибка.

Тестирование по принципу черного ящика не может охватить все аспекты испытываемого продукта. Современные программы слишком велики, чтобы реализовать имитацию щелчков по каждой кнопке, проверить все возможные варианты входных данных и выполнить все комбинации команд. Необходимость «белого» тестирования объясняется тем, что гораздо проще гарантировать тестовое покрытие множества неисправностей, если тесты будут написаны на уровне объекта или подсистемы. К тому же, зачастую «белые» тесты легче написать и автоматизировать, чем «черные». В этой работе делается акцент на методах тестирования путем применения «прозрачного ящика», поскольку программист может использовать эти методы еще во время разработки своей программы.

Единственный способ выявить ошибки в программе – провести ее тестирование. Одним из самых важных видов тестирования с точки зрения разработчика является блочное, или поэлементное [26]. Поэлементные тесты представляют собой программы, которые проверяют работоспособность класса или подсистемы. В идеальном варианте для каждой задачи низкого уровня должен существовать один или несколько тестов этого ранга.

Хорошо написанные тесты служат защитой во многих отношениях.

1) Они доказывают, что данная часть программы действительно работает должным образом. До тех пор, пока не будет получен код, который и в самом деле оправдывает существование тестируемого класса, его поведение можно считать неизвестным.
2) Поэлементные тесты первыми подают сигнал тревоги, если после недавнего изменения возник дефект.
3) Используемые как часть общего процесса разработки, они заставляют разработчика устранять проблемы с самого начала. Если какой-либо участок кода вообще не контролируется с помощью блочных тестов, то в случае возникновения проблемы ее источник можно смело искать в этом участке.

4) Поэлементные тесты позволяют испытать код до объединения с другим кодом.

5) Поэлементные тесты показывают пример применения созданного кода.

Увеличение количества написанных тестов влечет за собой расширение тестового покрытия кода и снижение вероятности невыявления ошибки.

Существуют разные методики написания поэлементных тестов [26]. Методология экстремального программирования предписывает своим сторонникам создавать тесты еще до написания кода. Теоретически предварительное написание тестов помогает четче сформулировать требования к компоненту и предложить систему показателей, которые могут быть использованы для определения момента завершения кода. Менее жесткий вариант состоит в проектировании тестов до кодирования, но в расчете на более позднюю их реализацию. В этом случае программист по-прежнему вынужден четко понимать требования, предъявляемые к модулю, но не обязан писать код, использующий еще несуществующие классы.

Код теста во многом зависит от типа используемой тестовой оболочки. В данной работе используется оболочка **CppUnit** – это программный набор инструментов для модульного тестирования программ на языках программирования C/C++ [13]. Система разрабатывается с 2000 г. и публикуется под GNU LGPL.

Оболочка запускает *тесты*, сгруппированные в *пакеты*. Результаты тестирования направляются в указанные фильтры, набор которых разнообразен: от простейшего подсчета пройденных либо не пройденных тестов до более продвинутых, поддерживающих XML-вывод, совместимый с дальнейшей интеграцией с системами отчетов.

Для того, чтобы понять, каким образом работает система тестирования `CppUnit`, рассмотрим, как спроектирован класс `MatrixTest`, основная задача которого – проведение блочного тестирования класса `Matrix`.

Объявление класса происходит в заголовочном файле (полный код класса приведен в приложении Б).

```
17 class MatrixTest : public CppUnit::TestFixture
```

Класс `MatrixTest` наследуется от класса `TestFixture`, который представляет собой логическую группу тестов.

Оболочка `CppUnit` выполняет группу тестов в виде некоторого набора (тестового комплекта), который должен содержать информацию о том, какие тесты подлежат выполнению, а какие нет (в отличие от класса `TestFixture`, который просто логически группирует тесты). Для этой цели `CppUnit` предоставляет класс `TestSuite`, который содержит в себе произвольный набор тестов. Обычная практика использования класса `TestSuite` состоит в следующем. В классы тестов (например такие, как `MatrixTest`), добавляется специальный статический метод.

```
1  public:
2    static CppUnit::Test* suite()
3    {
4      ...
5    }
```

После чего, посредством вызова этого метода, происходит добавление произвольного количества тестов в тестовый набор и запуск этого набора на выполнение. Как можно заметить, это довольно рутинная, повторяющаяся для каждого нового теста, задача. Поэтому `CppUnit` предоставляет набор вспомогательных макросов, которые призваны упростить процесс создания наборов тестов.

Поскольку разрабатывается класс, представляющий собой описание математического объекта (матрицы) в программе, для проверки его работоспособности видится естественным проведение как минимум следующих тестов:

- тест сложения матриц;
- тест умножения матриц;
- тест транспонирования матриц;
- тесты производительности.

В заголовочном файле класса `MatrixTest` происходит объявление перечисленных тестов.

```
19     CPPUNIT_TEST_SUITE( MatrixTest );
20       CPPUNIT_TEST( addition );
21       CPPUNIT_TEST( multiplication );
22       CPPUNIT_TEST( transposition );
```

```
23        CPPUNIT_TEST( acceleration );
24        CPPUNIT_TEST( multiple_threads_time );
25        CPPUNIT_TEST( threads_number_time );
26        CPPUNIT_TEST( matrix_types );
27      CPPUNIT_TEST_SUITE_END();
```

Это один из примеров использования вспомогательных макросов. Такой код осуществляет автоматическое создание тестового набора `MatrixTest` и добавление тестов в набор для последующего выполнения.

Далее происходит объявление соответствующих методов класса.

```
31    void crash_double_type();
32    void addition();
33    void multiplication();
34    void transposition();
35    void acceleration();
36    void multiple_threads_time();
37    void threads_number_time();
38    void matrix_types();
```

Тест `testCrashDoubleType` не был добавлен в набор, потому что демонстрирует появление ошибки округления, возникающей при использовании типа `double` и проводит к систематическому непрохождению тестового набора. Исходный код этого теста уже был проанализирован (см. листинг 1 и комментарии к нему).

Чтобы выполнить набор тестов и увидеть результаты, необходим *прогонщик тестов*. В оболочке `CppUnit` содержится несколько различных прогонщиков, которые действуют в различных средах (например, прогонщик `MFCRunner` предназначен для выполнения в программе, написанной с использованием библиотеки базовых классов Microsoft). Для текстовых сред используется прогонщик `TextRunner`. Ниже приводится код тела функции `main`, в котором происходит создание всех основных объектов тестового окружения.

```
10  int main( int argc, const char* argv[ ] )
11  {
12    CppUnit::TestResult controller;
13
14    CppUnit::TestResultCollector result;
15    controller.addListener( &result );
16
```

```
17    CppUnit::TextTestProgressListener progress;
18    controller.addListener( &progress );
19
20    CppUnit::TextUi::TestRunner runner;
21    runner.addTest( CppUnit::TestFactoryRegistry::getRegistry().makeTest() );
22
23    runner.run( controller, "" );
24
25    CppUnit::CompilerOutputter outputter( &result, std::cerr );
26    outputter.write();
27
28    return result.wasSuccessful() ? 0 : 1;
29  }
```

В строке 12 создается объект `controller` класса `TestResult`, представляющий собой статистику прохождения тестов. Другие вспомогательные классы будут подключаться именно к нему. В строке 14 создается сборщик статистики. В строке 17 – объект, следящий за ходом выполнения всего набора тестов. В строке 20 создается прогонщик тестов и запускается в строке 23. В строке 25 создается объект, отвечающий за печать результатов теста в совместимом с компилятором формате.

Рассмотрим структуру самих тестов, описанных в файле реализации. Для регистрации созданного набора тестов в системе используется такая строка.

```
16  CPPUNIT_TEST_SUITE_REGISTRATION( MatrixTest );
```

Для проверки утверждений и определения точного места, где они не выполняются, используется макрос `CPPUNIT_ASSERT`. Рассмотрим первый тест, где он применяется.

```
50  void MatrixTest::addition()
51  {
52    matrix_t A( this->matrix_size, this->matrix_size );
53    this->matrix_random_filler.fill( A );
54
55    matrix_t B( this->matrix_size, this->matrix_size );
56    this->matrix_random_filler.fill( B );
57
58    matrix_t C( this->matrix_size, this->matrix_size );
59    MatrixSummarizer matrix_summarizer;
60    matrix_summarizer.summarize( C, A, B );
61
62    matrix_t D = ( B + A );
```

```
63
64    CPPUNIT_ASSERT( C == D );
65  }
```

Это *тест, проверяющий корректность сложения матриц*. В строках 52–56 с помощью уже известных объектов вспомогательных классов происходит создание и заполнение операндов, участвующих в тесте. В строке 60 осуществляется вычисление результата сложения двух операндов сторонним классом и сохранение этого результата в переменной `C`. В строке 62 сложение выполняет сам класс `Matrix`, сохраняя результат в переменную `D`. В строке 64 осуществляется формальная проверка на равенство двух независимо полученных результатов. В этой точке теста *ожидается*, что значения этих результатов совпадут. Если окажется, что это не так, `CppUnit` сообщит о провале прохождения теста и укажет на эту точку, сообщив имя файла и номер строки.

Тест матричного умножения во многом схож с первым тестом. Его исходный код приведен ниже.

```
69  void MatrixTest::multiplication()
70  {
71    matrix_t A( this->matrix_size, this->matrix_size );
72    this->matrix_random_filler.fill( A );
73
74    matrix_t B( this->matrix_size, this->matrix_size );
75    this->matrix_random_filler.fill( B );
76
77    matrix_t C( this->matrix_size, this->matrix_size );
78    MatrixMultiplier matrix_multiplier;
79    matrix_multiplier.multiply( C, A, B );
80
81    matrix_t D = ( A * B );
82
83    CPPUNIT_ASSERT( C == D );
84
85    matrix_t E = A;
86    value_t value = 2;
87    matrix_multiplier.multiply( value, E );
88
89    matrix_t F( A * value );
90
91    CPPUNIT_ASSERT( F == E );
```

```
92 }
```

Единственное отличие состоит в дополнительной проверке в строках 85–91 работы операции умножения матрицы на число.

Тест транспонирования необходим для контроля правильности выполнения соответствующей матричной операции.

```
96  void MatrixTest::transposition()
97  {
98    matrix_t matrix( this->matrix_size, 2 * this->matrix_size );
99    this->matrix_random_filler.fill( matrix );
100
101   matrix_t matrix_transposed = matrix.transpose();
102
103   for( std::size_t iRow = 0; iRow < matrix.get_rows(); ++iRow ) {
104     for( std::size_t jColumn = 0; jColumn < matrix.get_columns(); ++jColumn ) {
105       CPPUNIT_ASSERT(
106         matrix( iRow, jColumn ) == matrix_transposed( jColumn, iRow )
107       );
108     }
109   }
110 }
```

Рассмотрим тест подробнее. В строках 98–99 создается случайно заполненная матрица. В строке 101 транспонированная матрица сохраняется в переменной `matrix_transposed`. В строках 103–109 в цикле происходит проверка результата транспонирования исходной матрицы. Смысл этой операции в том, чтобы убедиться в свойстве зеркальности операции транспонирования, а именно что (i, j)-й элемент исходной матрицы соответствует (j, i)-му элементу транспонированной.

Тестовый набор содержит дополнительно тесты производительности, осуществляющие замеры времени работы разработанного класса. Их исходный код приведен в приложении Б.

Таким образом, рассмотрены все блочные тесты, участвующие в проверке корректности работы разрабатываемого класса `Matrix`. Их регулярное прохождение будет доказывать правильность работы каждого участка кода. А в случае возникновения ошибки оболочка `CppUnit` позволит быстро локализовать и устранить проблему.

Выводы по главе два

Показано, каким образом модель Fork-Join используется для создания параллельной реализации метода эллипсоидов.

Приведены аргументированные доказательства необходимости обеспечения поддержки арифметики неограниченной точности с помощью библиотеки GMP.

Показаны и подробно прокомментированы особенности реализации класса матриц, обеспечивающего параллельное выполнение матричных операций.

Представлено тестовое окружение разработанного класса с обоснованием выбора тестовой оболочки.

3 Параллельная реализация метода эллипсоидов

В информатике параллельный алгоритм – алгоритм, который может быть реализован по частям на множестве различных вычислительных устройств с последующим объединением полученных результатов и получением корректного результата [17].

Параллельным алгоритмам противопоставляются традиционные алгоритмы – алгоритмы, выполняющиеся последовательно от начала и до конца.

Согласно описанию алгоритма 1, метод эллипсоидов относится к традиционному типу. Это означает, что логически алгоритм будет выполняться всегда последовательно в силу своей природы. Однако, если выполнять ресурсоемкие матричные операции параллельно с использованием уже разработанного класса, можно добиться ускорения работы метода вцелом.

3.1 Программная реализация метода эллипсоидов

Разработанная реализация метода эллипсоидов представляет собой шаблонный класс, параметризованный типом аргумента и размерностью евклидова пространства решаемой задачи (полный исходный код класса приведен в приложении В).

```
12 template< typename T, std::size_t Dimension >
13 class EllipsoidsMethod
```

Поскольку классический метод эллипсоидов Шора определен только для пространства $E^n,\ n \geq 2$ (см. описание метода эллипсоидов в главе один), в классе предусмотрена статическая проверка размерности пространства, указанного в качестве параметра шаблона.

```
15   static_assert( Dimension >= 2u, "Dimension of euclidean space is too small" );
```

В классе вводятся и используются сущности следующих типов.

```
17   typedef T value_t;
18   typedef Matrix< value_t > matrix_t;
19   typedef Point< value_t, Dimension > point_t;
20   typedef Constraint< value_t, Dimension > constraint_t;
21   typedef ConstraintList< value_t, Dimension > constraint_list_t;
```

Здесь тип, описанный в строке 17 – тип аргумента; 18 – матрица элементов типа аргумента; 19 – тип n-мерной точки пространства E^n; 20 – тип, представляющий *ограничение* задачи выпуклого программирования в виде пары функция–субградиент; 21 – список всех ограничений.

Единственная открытая функция `optimize` находит оптимальную точку по методу эллипсоидов и повторяет шаги алгоритма 1 из первой главы. Поскольку функция нетривиальна, рассмотрим ее подробнее. Начнем с аргументов функции.

```
37  point_t optimize( const constraint_t& objective,
38    const constraint_list_t& constraints,
39    const point_t& initial_point,
40    const value_t& ball_radius,
41    const value_t& accuracy,
42    std::size_t iteration_limit )
```

Описанные аргументы имеют следующие значения. Объект `objective` – это *целевая функция* задачи выпуклого программирования, значение которой *минимизируется*; `constraints` – *ограничения* решаемой задачи; `initial_point` – начальное приближение, с которого начинает расчет алгоритм; `ball_radius` – радиус шара, содержащего оптимальную точку; `accuracy` – требуемая точность решения задачи; `iteration_limit` – количество итераций, доступных методу для поиска (это способ выхода из алгоритма, если решение не сходится к оптимуму).

Работа функции – точное повторение шагов алгоритма. Поэтому для того, чтобы понять сущность выполняемых операций, написанных на языке программирования C++, уместными здесь будут обращения к математическим формулировкам алгоритма, и проведение соответствующей параллели с программным кодом.

Алгоритм начинается с инициализации. Строка

```
value_t reduction = ball_radius / ( Dimension + 1.0 );
```

обозначает то же, что и $h_k = \frac{R}{n+1}$. Далее в строке

```
point_t optimal_point = initial_point;
```

выполняется $x_k = x_0$, а в строках

```
const matrix_t identity = matrix_t::Type::identity( Dimension );
matrix_t inverse_transform_space( identity );
```

инициализируется матрица обратного преобразования пространства $B_k = E$.

Далее, происходит вычисление оракула

```
point_t subgradient = this->calculate_subgradient( objective,
  constraints,
  optimal_point
);
```

по формуле

$$g(x_k) = \begin{cases} g_0(x_k), & \text{если } \max\limits_{1 \le i \le m} f_i(x_k) \le 0, \\ g_{i^*}(x_k), & \text{если } \max\limits_{1 \le i \le m} f_i(x_k) = f_{i^*}(x_k) > 0. \end{cases} \tag{3.1}$$

После выполнения инициализирующих действий осуществляется подготовка к циклическим расчетам. Для этого в строках

```
std::size_t iteration = 0u;
bool stopping_criterion = false;
```

обнуляется счетчик итераций и значение критерия остановки выставляется в `false` для первого вхождения в цикл.

Все последующие расчеты выполняются циклически.

```
while( !stopping_criterion ) {

  );
```

Цикл начинается с вычисления вектора направления

```
point_t direction = inverse_transform_space.transpose() * subgradient;
direction *= ( 1.0 / direction.norm() );
```

по формуле $\xi_k = \dfrac{B_k^T g(x_k)}{||B_k^T g(x_k)||}$. После этого обновляется значение текущей точки

```
optimal_point -= ( inverse_transform_space * reduction ) * direction;
```

согласно выражению $x_{k+1} = x_k - h_k \cdot B_k \cdot \xi_k$. Затем пространство субградиентов сжимается.

```
const value_t beta = std::sqrt(
  ( Dimension - 1.0 ) / ( Dimension + 1.0 )
);
inverse_transform_space = inverse_transform_space * (
```

```
    identity + ( direction * direction.transpose() ) * ( beta - 1.0 )
  );
```

Для этого матрица обратного преобразования пространства умножается на оператор растяжения пространства с коэффициентом $\beta < 1$, т.е. вычисляется $B_{k+1} = B_k \cdot R_\beta(\xi_k)$. После этого выполняется

```
const value_t reduction_step = (
  Dimension / std::sqrt( std::pow( Dimension, 2 ) - 1.0 )
);

reduction *= reduction_step;
```

для вычисления $h_{k+1} = h_k \cdot r$, где $r = \frac{n}{\sqrt{n^2-1}}$. Наконец, обновляется значение текущего субградиента

```
subgradient = this->calculate_subgradient( objective,
  constraints,
  optimal_point
);
```

и счетчика итераций

```
++iteration;
```

для определения критерия останова. Критерий

```
stopping_criterion = (
  ( subgradient.norm() < accuracy ) || ( iteration >= iteration_limit )
);
```

проверяет достигнутую точность вычисления и следит за количеством выполненных итераций, не позволяя циклу выполняться больше, чем было затребовано. После этого тело цикла выполнятся снова до тех пор, пока не будет выполнено одно из условий критерия останова.

После выхода из цикла проверяется предварительное условие алгоритма метода эллипсоидов $||x^* - x_0|| < R$ – нахождение оптимальной точки в шаре радиуса R с центром в точке x_0. Для этого выполняется следующая инструкция.

```
bool precondition = (
  static_cast< point_t >(
    optimal_point - initial_point
  ).norm() <= ball_radius
);
```

От результата этой проверки будет зависеть возвращаемый результат. Если найденная оптимальная точка находится внутри шара, то возвращается ее значение. В противном случае функция возвращает значение начальной точки. Это поведение реализовано с помощью тернарного оператора.

```
return precondition ? optimal_point : initial_point;
```

Функция вычисления субградиента, код которой представлен ниже, логически работает строго по формуле вычисления оракула (3.1).

```
point_t calculate_subgradient( const constraint_t& objective,
  const constraint_list_t& constraints,
  const point_t& point ) const
{
  auto max = std::max_element( constraints.begin(), constraints.end(),
    [ & ]( const constraint_t& lhs, const constraint_t& rhs ) -> bool {
      return ( lhs.function( point ) < rhs.function( point ) );
    } );

  point_t subgradient;
  if( ( *max ).function( point ) <= value_t( 0 ) ) {
    subgradient = objective.subgradient( point );
  }
  else {
    subgradient = ( *max ).subgradient( point );
  }

  return subgradient;
}
```

Таким образом, было полностью рассмотрено внутреннее устройство класса `EllipsoidsMethod`, прокомментированы основные особенности реализации.

3.2 Вычислительный эксперимент

Для доказательства правильности работы разработанной реализации метода эллипсоидов, выполним оптимизацию какой-либо функции в некоторой заданной области, значение минимума которой заранее известно.

Рассмотрим задачу минимизации

$$f_0(x_1, x_2) = x_1^2 + (x_2 - 2)^2 \to \min \tag{3.2}$$

при ограничениях

$$\begin{cases} f_1(x_1, x_2) &= x_1^2 + x_2^2 - 9; \\ f_2(x_1, x_2) &= x_1^2 + (x_2 - 4)^2 - 9. \end{cases} \tag{3.3}$$

Графическое изображение рассматриваемой задачи выпуклого программирования приведено на рисунке 8. Целевая функция представляет собой параболоид с вершиной в точке $(0, 2, 0)$ (показан красным; вершина параболоида и есть минимальное значение целевой функции в заданной области), а область поиска представляет собой пересечение двух ограничений – шаров радиуса 3 с центрами в точках $(0, 0, 0)$ и $(0, 4, 0)$ соответственно (показаны синим).

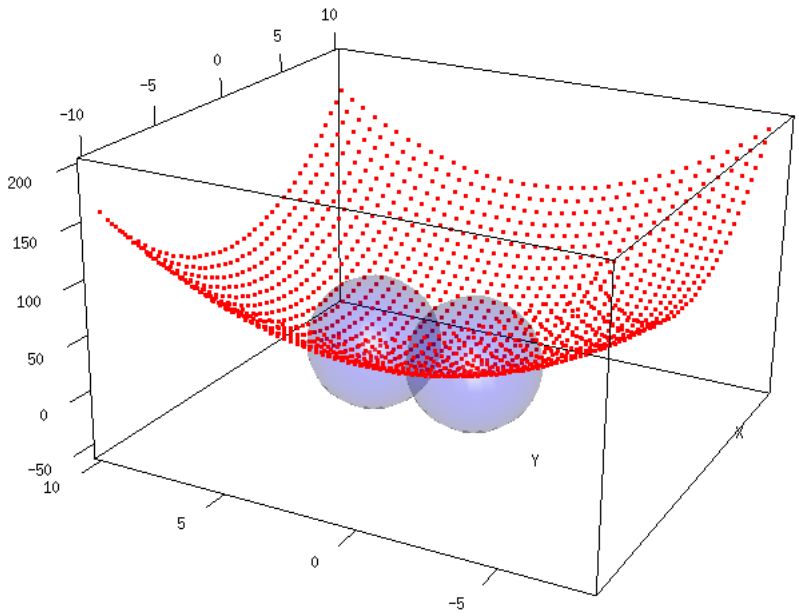

Рисунок 8 — Графическое представление задачи оптимизации

Создадим тест `UnitCPP`, в котором сделаем следующие объявления.

```
typedef mpf_class value_t;
const std::size_t Dimension = 2;
typedef Point< value_t, Dimension > point_t;
```

Для проведения расчетов был выбран тип `mpf_class` библиотеки GMP, представляющий в программе числа с плавающей запятой неограниченной точности. Константа `Dimension` представляет размерность евклидова пространства рассматриваемой задачи. Соответственно, тип `point_t` введен для представления двумерной точки $x \in E^2$.

Объявим в тесте функции $f_\nu(x_1, x_2),\ \nu = \overline{0,2}$, указанные в постановке задачи (3.2)–(3.3).

```
/**
 * f0 = x1^2 + ( x2 - 2 )^2
 */
auto objective_function = []( const point_t& x ) -> value_t {
  return ( std::pow( x[0], 2 ) + std::pow( x[1] - 2, 2 ) );
};

/**
 * f1 = x1^2 + x2^2 - 9
 */
auto constraint_1 = []( const point_t& x ) -> value_t {
  return ( std::pow( x[0], 2 ) + std::pow( x[1], 2 ) - 9 );
};

/**
 * f2 = x1^2 + ( x2 - 4 )^2 - 9
 */
auto constraint_2 = []( const point_t& x ) -> value_t {
  return ( std::pow( x[0], 2 ) + std::pow( x[1] - 4, 2 ) - 9 );
};
```

Вычислим субградиенты функций $f_\nu(x_1, x_2),\ \nu = \overline{0,2}$. Для рассматриваемой задачи они будут совпадать с градиентами.

$$\begin{cases} g_0(x_1, x_2) &= (2x_1, 2(x_2 - 2)); \\ g_1(x_1, x_2) &= (2x_1, 2x_2); \\ g_2(x_1, x_2) &= (2x_1, 2(x_2 - 4)) \end{cases} \tag{3.4}$$

Представим субградиенты функций в тестовом методе.

```
/**
 * g0 = ( 2 * x1, 2 * ( x2 - 2 ) )
 */
auto objective_subgradient = []( const point_t& x ) -> point_t {
  return point_t{ 2 * x[0], 2 * ( x[1] - 2 ) };
};

/**
 * g1 = ( 2 * x1, 2 * x2 )
 */
auto subgradient_1 = []( const point_t& x ) -> point_t {
  return point_t{ 2 * x[0], 2 * x[1] };
```

```
};

/**
 * g2 = ( 2 * x1, 2 * ( x2 - 4 ) )
 */
auto subgradient_2 = []( const point_t& x ) -> point_t {
  return point_t{ 2 * x[0], 2 * ( x[1] - 4 ) };
};
```

В соответствии с описанием функции `optimize` (см. подраздел 3.1), все участвующие в расчетах функции и субградиенты необходимо «упаковать» в сущности типа `Constraint`. Для целевой функции представим объект `objective`.

```
Constraint< value_t, Dimension > objective( objective_function,
  objective_subgradient
);
```

Для ограничений создадим отдельный список и добавим оставшиеся функции.

```
ConstraintList< value_t, Dimension > constraints;
constraints.add( constraint_1, subgradient_1 );
constraints.add( constraint_2, subgradient_2 );
```

В качестве начальной выберем точку $x_0 = (1, 2)$, радиус шара (начальной области поиска) положим равным $R = 10$, зададим критерий остановки по точности $\varepsilon = 10^{-10}$ и максимально допустимое число итераций $Q = 1000$. Создадим описанные параметры в тестовом методе.

```
const point_t initial_point{ 1.0, 2.0 };
const value_t ball_radius = 10.0;
const value_t epsilon = 10e-10;
const std::size_t iteration_limit = 1000;
```

Запустим алгоритм метода эллипсоидов для задачи (3.2)–(3.3) с указанными параметрами, сохранив оптимальное значение в объекте `point`.

```
EllipsoidsMethod< value_t, Dimension > ellipsoids_method;
point_t point = ellipsoids_method.optimize( objective,
  constraints,
  initial_point,
  ball_radius,
  epsilon,
  iteration_limit
);
```

Наконец, распечатаем найденное значение с большой точностью, используя вспомогательный класс.

```
std::cout << "Iterations: " << ellipsoids_method.get_iterations();

MatrixPrinter matrix_printer( &std::cout );
matrix_printer.set_precision( 20 );
matrix_printer.print( point );
```

Для рассматриваемой задачи алгоритм метода эллипсоидов выдал результат, согласующийся с ожидаемым:

$$point = \begin{pmatrix} 0.00000000011922745523 \\ 2.00000000033459867651 \end{pmatrix}.$$

Вычислительный процесс сошелся к решению за 160 итераций, т.е. остановка счета произошла по достижении заданной точности в 9 знаков после запятой.

3.3 Решение задачи оптимизации большой размерности

Продемонстрируем использование созданной реализации алгоритма метода эллипсоидов для решения оптимизационной задачи большой размерности.

Рассмотрим задачу минимизации

$$f_0(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i^2 \to \min, \tag{3.5}$$

при ограничениях

$$f_m(x_1, x_2, \ldots, x_n) = \sum_{i=1, i \neq m}^{n} x_i^2 + (x_m - \alpha/2)^2 - \alpha^2, \; m = \overline{1, n}. \tag{3.6}$$

Все функции f_ν, $\nu = \overline{0, n}$ определены на E^n. Целевая функция (3.5) представляет собой n-мерную параболу с вершиной в нуле, а каждое из ограничений (3.6) – это n-мерный шар радиуса α, смещенный по одной из осей на $\alpha/2$.

Ясно, что рассматриваемая задача оптимизации при данных ограничениях имеет оптимум в начале координат. Этот пример рассматривается исключительно как тестовый. Выбор именно такой формы целевой функции и ограничений обуславливается простотой их процедурной генерации.

Решим задачу (3.5)–(3.6) численно. Рассмотрим случай $n = 100,\ \alpha = 1$. Субградиент целевой функции

$$g_0(x_1, x_2, \ldots, x_n) = (2x_1, 2x_2, \ldots, 2x_n),$$

субградиенты ограничений

$$g_m(x_1, x_2, \ldots, x_n) = (2x_1, \ldots, 2x_{m-1}, 2(x_m - \alpha/2), 2x_{m+1}, \ldots, 2x_n).$$

Положим начальную точку $x_0 = (1/2, 0, 0, \ldots, 0)$, радиус шара $R = 10$, критерий остановки по точности $\varepsilon = 10^{-10}$, ограничение на количество итераций $Q = 1000$, и произведем вычисления аналогично примеру из подраздела 3.2.

По завершении процесса вычисления и печати полученного результата, имеем значение оптимальной точки $x^* = (0, 0, \ldots, 0)$. Вычислительный процесс сошелся к решению за 403 итерации, т.е. останов произошел по достигнутой точности.

Покажем, что разработанная реализация алгоритма метода эллипсоидов решает такую задачу оптимизации быстрее, благодаря использованию многопоточности. Для этого проведем вычислительный эксперимент.

Сначала инструкцией

```
const std::size_t num_threads = 1;
Matrix< value_t >::get_parallel_handler()->set_direct_parallel_policy(
  num_threads );
```

запретим программной реализации использование доступного аппаратного параллелизма. Запустим решение задачи (3.5)–(3.6) и замерим время ее выполнения.

Затем активируем параллелизм командой

```
Matrix< value_t >::get_parallel_handler()->set_auto_parallel_policy();
```

для использования в расчетах всех доступных аппаратно поддерживаемых потоков конкретной машины, и снова произведем расчеты и замерим время. После выполнения такой последовательности действий имеем следующие результаты.

При тестовом запуске с использованием одного потока время вычисления составило $T_s = 76.889$, с использованием двух потоков $T_m^2 = 44.509$, с использованием восьми потоков $T_m^8 = 28.1456$. Графическая интерпретация полученных результатов представлена на рисунке 9.

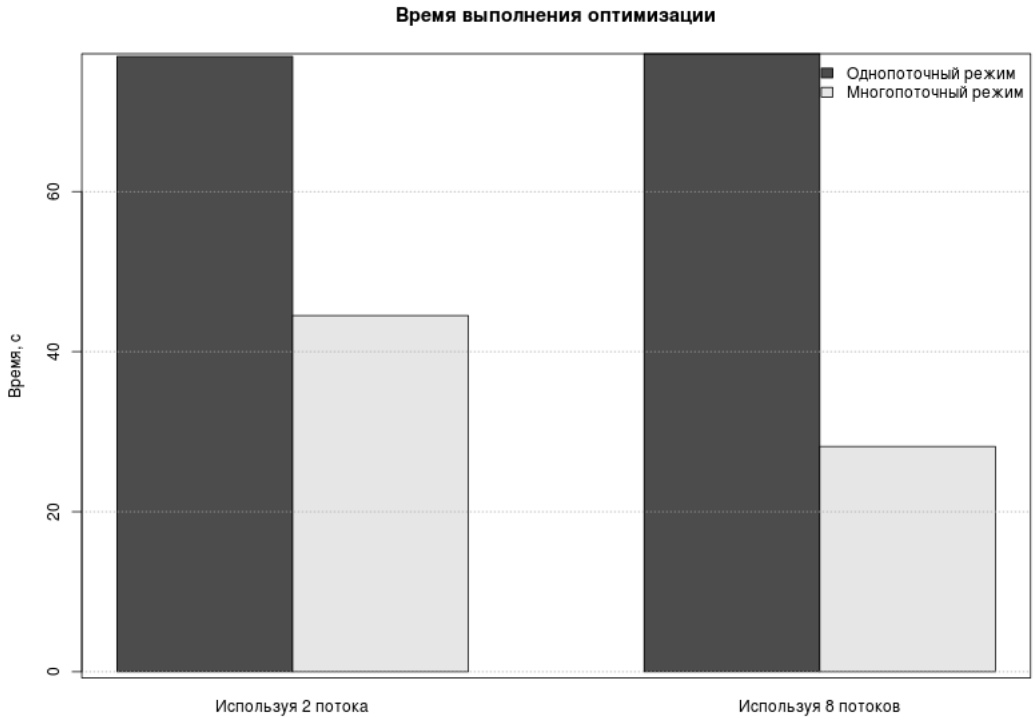

Рисунок 9 — Время выполнения оптимизации

Относительные коэффициенты ускорения для двух экспериментов составляют $k_1 = T_s/T_m^2 = 76.889/44.509 = 1.727$ и $k_2 = T_s/T_m^8 = 76.889/28.1456 = 2.744$ соответственно. Поскольку разработанная реализация метода эллипсоидов опирается на класс матриц, поддерживающих распараллеливание операций, отсутствие линейного роста коэффициентов ускорения объясняется отсутствием такового у класса матриц. Так, увеличение числа потоков до 2 привело к уменьшению времени выполнения в 1.727 раз, в то время как использование 8 одновременно выполняющихся потоков уменьшает общее

время выполнения алгоритма лишь в 2.744 раза. Экстраполируя рассуждения, предположим, что с большой вероятностью дальнейшее увеличение числа параллельных потоков будет сокращать время все меньше. Если и дальше продолжать увеличивать их количество, то время выполнения начнет не уменьшаться, а *расти* и может даже превысить время однопоточной версии. Как уже было сказано, это будет происходить из-за дополнительных *накладных расходов*, связанных с организацией параллелизма.

Тем не менее, возвращаясь к формулировке гипотезы 1 о соотношении времени, видим, что выдвинутые предположения **оправдались**: для разработанной реализации алгоритма метода эллипсоидов справедливо равенство

$$F(t) = kf(t),$$

где $f(t)$ – это время работы алгоритма, выполняемого в однопоточном режиме, $F(t)$ – время работы параллельной реализации метода, k – коэффициент ускорения и $k > 1$. Это справедливо для достаточно большой задачи оптимизации, в ходе решения которой возникает необходимость оперирования матрицами размерности 100×100.

Выводы по главе три

Разработана программная реализация алгоритма метода эллипсоидов. В главе даны подробные комментарии и объяснения относительно особенностей внутреннего устройства созданных классов.

Показана работоспособность разработанного ПО путем численного решения задачи выпуклого программирования.

Продемонстрировано решение задачи оптимизации большой размерности.

В ходе численных экспериментов установлено, что разработанное ПО позволяет эффективно использовать ресурсы вычислительной системы и дает ускорение времени выполнения, большее 1.

Заключение

В работе представлена параллельная реализация алгоритма метода эллипсоидов.

В работе решены следующие **задачи**:

- операции классического алгоритма метода эллипсоидов исследованы на вычислительную сложность;
- разработана программная реализация алгоритма с распараллеливанием наиболее длительных по времени операций;
- обеспечена поддержка арифметики расширенной и произвольной точности;
- продемонстрировано использование разработанного ПО для решения задачи оптимизации большой размерности;
- разработанный код проверен и протестирован.

На основе анализа результатов вычислительных экспериментов можно сделать следующие выводы:

- параллельная реализация метода эллипсоидов позволяет за то же время решать задачи оптимизации большей размерности;
- разработанное ПО, поддерживающее параллельное выполнение операций, эффективнее использует ресурсы современной вычислительной системы;
- для задач оптимизации большой размерности разработанная реализация выполняется быстрее однопоточного алгоритма и имеет ускорение, превышающее 1.

В качестве направлений дальнейших исследований следует рассматривать следующие:

1) повышение эффективности работы с типами данных повышенной точности;
2) применение последних алгоритмических разработок в области параллельного умножения матриц;
3) использование модификаций метода эллипсоидов для решения одномерных задач оптимизации.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД КЛАССА MATRIX

Листинг А.1 — Файл «Matrix.h»

```
#ifndef MATRIX
#define MATRIX

#include <assert.h>
#include <valarray>

#include "ParallelHandler.h"


/**
 *  Represents matrices.
 */
template< typename T >
class Matrix
{
private:

  typedef T value_t;
  typedef Matrix< value_t > my_t;
  typedef std::size_t index_t;



public:

  /**
   * A constructor.
   * Construct a rows-by-columns Matrix filled with the given value.
   */
  Matrix( index_t rows, index_t columns, const value_t& value = value_t( 0 ) )
    : values( value, rows * columns )
    , rows( rows )
    , columns( columns )
    , is_transposed( false )
  {
    assert( rows > 0 );
    assert( columns > 0 );
  }
```

```
/**
 * A copy constructor.
 * Construct a copy of other Matrix object.
 */
Matrix( const my_t& rhs )
  : values( rhs.values )
  , rows( rhs.rows )
  , columns( rhs.columns )
  , is_transposed( rhs.is_transposed )
{

}



/**
 * A move constructor.
 * Construct a copy using resources of temporary Matrix object.
 */
Matrix( my_t&& rhs )
  : values( std::move( rhs.values ) )
  , rows( rhs.rows )
  , columns( rhs.columns )
  , is_transposed( rhs.is_transposed )
{

}



/**
 * An accessor.
 * Retrieve the reference.
 */
value_t& operator() ( const index_t& row, const index_t& column )
{
  assert( row < this->get_rows() );
  assert( column < this->get_columns() );

  return this->get( row, column );
}
```

```
/**
 * A const accessor.
 * Retrieve the const reference.
 */
const value_t& operator() ( const index_t& row, const index_t& column ) const
{
  assert( row < this->get_rows() );
  assert( column < this->get_columns() );

  return this->get( row, column );
}



/**
 * A copy-based assignment operator.
 * Make the matrix equal to other Matrix object.
 */
my_t& operator= ( const my_t& rhs )
{
  if( this == &rhs ) {
    return *this;
  }

  this->values = rhs.values;
  this->rows = rhs.rows;
  this->columns = rhs.columns;
  this->is_transposed = rhs.is_transposed;

  return *this;
}



/**
 * A move-based assignment operator.
 * Make the matrix the owner of the resources of the temporary Matrix object.
 */
my_t& operator= ( my_t&& rhs )
{
  if (this == &rhs) {
    return *this;
  }

  this->values = std::move( rhs.values );
  this->rows = rhs.rows;
```

```
  this->columns = rhs.columns;
  this->is_transposed = rhs.is_transposed;

  return *this;
}



/**
 * An addition operator.
 * Retrieve the Matrix each element of which is the sum of the corresponding
 * values of this and rhs Matrix objects.
 */
my_t operator+ ( const my_t& rhs ) const
{
  assert( this->get_rows() == rhs.get_rows() );
  assert( this->get_columns() == rhs.get_columns() );

  my_t result( this->get_rows(), this->get_columns() );

  auto summarize = [ this, &rhs, &result ]( const index_t& row_start,
    const index_t& row_end ) -> void
  {
    for( index_t row = row_start; row < row_end; ++row ) {
      for( index_t column = 0; column < this->get_columns(); ++column ) {
        result.get( row, column ) = (
          this->get( row, column ) + rhs.get( row, column )
        );
      }
    }
  };

  const index_t first = 0;
  const index_t last = this->get_rows();
  my_t::get_parallel_handler()->parallel_for( first, last, summarize );

  return result;
}



/**
 * A subtraction operator.
 * Retrieve the Matrix each element of which is the difference of the
 * corresponding values of this and rhs Matrix objects.
 */
```

```
my_t operator- ( const my_t& rhs ) const
{
  assert( this->get_rows() == rhs.get_rows() );
  assert( this->get_columns() == rhs.get_columns() );

  my_t result( this->get_rows(), this->get_columns() );

  auto subtract = [ this, &rhs, &result ]( const index_t& row_start,
    const index_t& row_end ) -> void
  {
    for( index_t row = row_start; row < row_end; ++row ) {
      for( index_t column = 0; column < this->get_columns(); ++column ) {
        result.get( row, column ) = (
          this->get( row, column ) - rhs.get( row, column )
        );
      }
    }
  };

  const index_t first = 0;
  const index_t last = this->get_rows();
  my_t::get_parallel_handler()->parallel_for( first, last, subtract );

  return result;
}



/**
 * A multiplication operator.
 * Retrieve the Matrix which is equal to result of matrix multiplication
 * of this and rhs Matrix objects.
 */
my_t operator* ( const my_t& rhs ) const
{
  assert( this->get_columns() == rhs.get_rows() );

  my_t result( this->get_rows(), rhs.get_columns() );

  auto multiply = [ this, &rhs, &result ]( const index_t& row_start,
    const index_t& row_end ) -> void
  {
    for( index_t row = row_start; row < row_end; ++row ) {
      for( index_t column = 0; column < rhs.get_columns(); ++column ) {
        for( index_t sIndex = 0; sIndex < this->get_columns(); ++sIndex ) {
          result.get( row, column ) += (
```

```
            this->get( row, sIndex ) * rhs.get( sIndex, column )
          );
        }
      }
    }
  };

  const index_t first = 0;
  const index_t last = this->get_rows();
  my_t::get_parallel_handler()->parallel_for( first, last, multiply );

  return result;
}



/**
 * A multiplication operator.
 * Multiply each element of the Matrix by value.
 */
my_t operator* ( const T& value ) const
{
  my_t result( *this );

  auto multiply = [ &result, &value ]( const index_t& row_start,
    const index_t& row_end ) -> void
  {
    for( index_t row = row_start; row < row_end; ++row ) {
      for( index_t column = 0; column < result.get_columns(); ++column ) {
        result.get( row, column ) *= value;
      }
    }
  };

  const index_t first = 0;
  const index_t last = result.get_rows();
  my_t::get_parallel_handler()->parallel_for( first, last, multiply );

  return result;
}



/**
 * A comparison operator.
 * Check if each element of the matrix is equal to the corresponding
```

```
 * element of the other Matrix object.
 */
bool operator== ( const my_t& rhs ) const
{
  if ( &rhs == this ) {
    return true;
  }

  bool rows_mismatch = ( this->get_rows() != rhs.get_rows() );
  bool columns_mismatch = ( this->get_columns() != rhs.get_columns() );

  if ( rows_mismatch || columns_mismatch ) {
    return false;
  }

  std::valarray< bool > comparison_result = ( this->values == rhs.values );
  bool is_equal = ( comparison_result.min() == true );

  return is_equal;
}



/**
 * A comparison operator.
 * Check if the matrix is not equal to the other Matrix object.
 */
bool operator!= ( const my_t& rhs ) const
{
  return !( *this == rhs );
}



/**
 * A transposition.
 * Swap rows and columns.
 */
my_t transpose() const
{
  my_t copy( *this );
  copy.is_transposed = !this->is_transposed;

  return copy;
}
```

```
/**
 * A number of rows.
 * Retrieve the number of rows of the matrix.
 */
std::size_t get_rows() const
{
  return ( this->is_transposed ? this->columns : this->rows );
}



/**
 * A number of columns.
 * Retrieve the number of columns of the matrix.
 */
std::size_t get_columns() const
{
  return ( this->is_transposed ? this->rows : this->columns );
}



/**
 * Retrieve an instance of parallel execution controller.
 */
static ParallelHandler* get_parallel_handler()
{
  static ParallelHandler parallel_handler;

  return &parallel_handler;
}



/**
 * Represents types of matrices.
 */
class Type
{
public:

  /**
   * An identity matrix.
   * Retrieve an identity matrix.
```

```
     */
    static Matrix< T > identity( index_t size )
    {
      Matrix< T > identity_matrix( size, size );

      for( index_t index = 0; index < size; ++index ) {
        identity_matrix( index, index ) = value_t( 1 );
      }

      return identity_matrix;
    }
  };



private:

  /**
   * A private accessor.
   * Retrieve the reference by index.
   */
  value_t& get( index_t row, index_t column )
  {
    if( this->is_transposed ) {
      std::swap( row, column );
    }

    index_t index = ( row * this->columns + column );

    return this->values[ index ];
  }



  /**
   * A const private accessor.
   * Retrieve the const reference by index.
   */
  const value_t& get( index_t row, index_t column ) const
  {
    if( this->is_transposed ) {
      std::swap( row, column );
    }

    index_t index = ( row * this->columns + column );
```

```
    return this->values[ index ];
  }



  std::valarray< value_t > values;  /**< A storage for matrix elements. */
  index_t rows;                     /**< A number of rows. */
  index_t columns;                  /**< A number of columns. */

  bool is_transposed;               /**< A transposition flag. */
};



#endif // MATRIX
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД КЛАССА MATRIXTEST

Листинг Б.1 — Файл «MatrixTest.h»

```
#ifndef MATRIX_TEST
#define MATRIX_TEST

#include <gmpxx.h>

#include <cppunit/extensions/HelperMacros.h>
#include <cppunit/TestFixture.h>

#include "Matrix.h"

#include "HelperClasses/MatrixRandomFiller.h"
#include "HelperClasses/Statistics.h"
#include "HelperClasses/TimeMeasurer.h"



class MatrixTest : public CppUnit::TestFixture
{
    CPPUNIT_TEST_SUITE( MatrixTest );
      CPPUNIT_TEST( addition );
      CPPUNIT_TEST( multiplication );
      CPPUNIT_TEST( transposition );
      CPPUNIT_TEST( acceleration );
      CPPUNIT_TEST( multiple_threads_time );
      CPPUNIT_TEST( threads_number_time );
      CPPUNIT_TEST( matrix_types );
    CPPUNIT_TEST_SUITE_END();

public:

  void crash_double_type();
  void addition();
  void multiplication();
  void transposition();
  void acceleration();
  void multiple_threads_time();
  void threads_number_time();
  void matrix_types();
```

```
private:

  template< typename Function >
  double calculate_average_time( Function&& function ) {
    TimeMeasurer time_measurer;
    std::valarray< double > durations( this->inner_loop_iterations_num );
    for( std::size_t duration = 0; duration < durations.size(); ++duration ) {
      time_measurer.start();
      function();
      time_measurer.end();
      durations[ duration ] = time_measurer.get_duration_in_seconds();
    }

    return durations.sum() / durations.size();
  }



  typedef mpf_class value_t;
  typedef Matrix< value_t > matrix_t;

  std::size_t matrix_size = 50;
  std::size_t initial_size = 10;
  std::size_t size_step = 10;
  std::size_t iterations_num = 0;
  std::size_t inner_loop_iterations_num = 1;

  Statistics statistics;
  MatrixRandomFiller matrix_random_filler;
};



#endif // MATRIX_TEST
```

Листинг Б.2 — Файл «MatrixTest.cpp»

```
#include "MatrixTest.h"

#include <valarray>

#include <cppunit/TestCaller.h>
#include <cppunit/TestSuite.h>

#include "HelperClasses/MatrixPrinter.h"
```

```
#include "HelperClasses/Statistics.h"

#include "HelperClasses/MatrixSummarizer.h"
#include "HelperClasses/MatrixMultiplier.h"



CPPUNIT_TEST_SUITE_REGISTRATION( MatrixTest );



void MatrixTest::crash_double_type()
{
  std::size_t rows = 2;
  std::size_t columns = 2;

  Matrix< double > A( rows, columns );
  Matrix< double > B( rows, columns );

  MatrixRandomFiller matrix_random_filler;
  matrix_random_filler.fill( A );
  matrix_random_filler.fill( B );

  MatrixPrinter matrix_printer( &std::cout );
  matrix_printer.set_precision( 20 );
  matrix_printer.print( A );
  matrix_printer.print( B );

  Matrix< double > C( rows, columns );
  MatrixMultiplier matrix_multiplier;
  matrix_multiplier.multiply( C, A, B );

  Matrix< double > D( rows, columns );
  D = A * B;

  matrix_printer.print( C );
  matrix_printer.print( D );
}



void MatrixTest::addition()
{
  matrix_t A( this->matrix_size, this->matrix_size );
  this->matrix_random_filler.fill( A );
```

```
  matrix_t B( this->matrix_size, this->matrix_size );
  this->matrix_random_filler.fill( B );

  matrix_t C( this->matrix_size, this->matrix_size );
  MatrixSummarizer matrix_summarizer;
  matrix_summarizer.summarize( C, A, B );

  matrix_t D = ( B + A );

  CPPUNIT_ASSERT( C == D );
}



void MatrixTest::multiplication()
{
  matrix_t A( this->matrix_size, this->matrix_size );
  this->matrix_random_filler.fill( A );

  matrix_t B( this->matrix_size, this->matrix_size );
  this->matrix_random_filler.fill( B );

  matrix_t C( this->matrix_size, this->matrix_size );
  MatrixMultiplier matrix_multiplier;
  matrix_multiplier.multiply( C, A, B );

  matrix_t D = ( A * B );

  CPPUNIT_ASSERT( C == D );

  matrix_t E = A;
  value_t value = 2;
  matrix_multiplier.multiply( value, E );

  matrix_t F( A * value );

  CPPUNIT_ASSERT( F == E );
}



void MatrixTest::transposition()
{
  matrix_t matrix( this->matrix_size, 2 * this->matrix_size );
  this->matrix_random_filler.fill( matrix );
```

```
  matrix_t matrix_transposed = matrix.transpose();

  for( std::size_t iRow = 0; iRow < matrix.get_rows(); ++iRow ) {
    for( std::size_t jColumn = 0; jColumn < matrix.get_columns(); ++jColumn ) {
      CPPUNIT_ASSERT(
        matrix( iRow, jColumn ) == matrix_transposed( jColumn, iRow )
      );
    }
  }
}



void MatrixTest::acceleration()
{
  std::size_t size = this->initial_size;
  for( std::size_t iteration = 0; iteration < iterations_num; ++iteration ) {
    matrix_t lhs( size, size );
    this->matrix_random_filler.fill( lhs );

    matrix_t rhs( size, size );
    this->matrix_random_filler.fill( rhs );

    matrix_t result( size, size );

    auto single_thread_summarizing = [ & ]() {
      std::size_t nthreads = 1;
      matrix_t::get_parallel_handler()->set_direct_parallel_policy( nthreads );
      result = ( lhs + rhs );
    };

    auto multiple_thread_summarizing = [ & ]() {
      matrix_t::get_parallel_handler()->set_auto_parallel_policy();
      result = ( lhs + rhs );
    };

    auto single_thread_multiplication = [ & ]() {
      std::size_t nthreads = 1;
      matrix_t::get_parallel_handler()->set_direct_parallel_policy( nthreads );
      result = ( lhs * rhs );
    };

    auto multiple_thread_multiplication = [ & ]() {
      matrix_t::get_parallel_handler()->set_auto_parallel_policy();
      result = ( lhs * rhs );
    };
```

```
    this->statistics( size,
      Statistics::Type::Addition ).set_single_thread_time(
        this->calculate_average_time( single_thread_summarizing )
    );

    this->statistics( size,
      Statistics::Type::Addition ).set_multiple_threads_time(
        this->calculate_average_time( multiple_thread_summarizing )
    );

    this->statistics( size,
      Statistics::Type::Multiplication ).set_single_thread_time(
        this->calculate_average_time( single_thread_multiplication )
    );

    this->statistics( size,
      Statistics::Type::Multiplication ).set_multiple_threads_time(
        this->calculate_average_time( multiple_thread_multiplication )
    );

    size = this->initial_size * (this->size_step + 10 * iteration);
  }

  const std::string file = "Acceleration statistics";
  this->statistics.save( file );
}



void MatrixTest::multiple_threads_time()
{
  this->inner_loop_iterations_num = 1;
  this->initial_size = 500;
  this->size_step = 500;
  this->iterations_num = 0;

  std::size_t size = this->initial_size;
  for( std::size_t iteration = 0;
    iteration < this->iterations_num; ++iteration )
  {
    matrix_t lhs( size, size );
    this->matrix_random_filler.fill( lhs );

    matrix_t rhs( size, size );
    this->matrix_random_filler.fill( rhs );
```

```
    matrix_t multiple_thread_result( size, size );

    auto multiple_thread_summarizing = [ & ]() {
      multiple_thread_result = ( lhs + rhs );
    };

    auto multiple_thread_multiplication = [ & ]() {
      multiple_thread_result = ( lhs * rhs );
    };

    this->statistics( size,
      Statistics::Type::Addition ).set_multiple_threads_time(
        this->calculate_average_time( multiple_thread_summarizing )
    );

    this->statistics( size,
      Statistics::Type::Multiplication ).set_multiple_threads_time(
        this->calculate_average_time( multiple_thread_multiplication )
    );

    size += this->size_step;
  }

  const std::string file = "Multithreading statistics";
  this->statistics.save( file );
}



void MatrixTest::threads_number_time()
{
  const std::size_t matrix_size = 500;
  this->inner_loop_iterations_num = 0;

  matrix_t lhs( matrix_size, matrix_size );
  this->matrix_random_filler.fill( lhs );

  matrix_t rhs( matrix_size, matrix_size );
  this->matrix_random_filler.fill( rhs );

  matrix_t result( matrix_size, matrix_size );

  auto multiple_thread_multiplication = [ & ]() {
    result = ( lhs * rhs );
  };
```

```
  double time = this->calculate_average_time( multiple_thread_multiplication );

  const std::string file = "Time (using 2 threads)";
  std::ofstream fstream( file );

  fstream << time;
}



void MatrixTest::matrix_types()
{
  Matrix< int > E( 3, 3 );
  E( 0, 0 ) = 1;
  E( 1, 1 ) = 1;
  E( 2, 2 ) = 1;
  CPPUNIT_ASSERT( E == Matrix< int >::Type::identity( 3 ) );
}
```

ПРИЛОЖЕНИЕ В

ИСХОДНЫЙ КОД КЛАССА ELLIPSOIDSMETHOD

Листинг В.1 — Файл «EllipsoidsMethod.h»

```
#ifndef ELLIPSOIDS_METHOD
#define ELLIPSOIDS_METHOD

#include <algorithm>

#include "ConstraintList.h"
#include "Matrix.h"

/**
 * An implementation of Shor's ellipsoids method.
 */
template< typename T, std::size_t Dimension >
class EllipsoidsMethod
{
  static_assert( Dimension >= 2u, "Dimension of euclidean space is too small" );

  typedef T value_t;
  typedef Matrix< value_t > matrix_t;
  typedef Point< value_t, Dimension > point_t;
  typedef Constraint< value_t, Dimension > constraint_t;
  typedef ConstraintList< value_t, Dimension > constraint_list_t;



public:

  /**
   * An implementation of ellipsoids method algorithm.
   * @param objective a function to be optimized
   * @param constraints a list of constraints of convex programming problem
   * @param initial_point a point from where the optimization process starts
   * @param ball_radius a radius of a ball which locates the optimal point
   * @param accuracy a stopping criterion for accuracy
   * @param iteration_limit a stopping criterion for number of iterations
   * @return The optimal point if success or initial point if failure
   */
  point_t optimize( const constraint_t& objective,
    const constraint_list_t& constraints,
    const point_t& initial_point,
```

```
  const value_t& ball_radius,
  const value_t& accuracy,
  std::size_t iteration_limit )
{
  value_t reduction = ball_radius / ( Dimension + 1.0 );
  point_t optimal_point = initial_point;

  const matrix_t identity = matrix_t::Type::identity( Dimension );
  matrix_t inverse_transform_space( identity );

  point_t subgradient = this->calculate_subgradient( objective,
    constraints,
    optimal_point
  );

  std::size_t iteration = 0u;
  bool stopping_criterion = false;
  while( !stopping_criterion ) {
    point_t direction = inverse_transform_space.transpose() * subgradient;
    direction *= ( 1.0 / direction.norm() );

    optimal_point -= ( inverse_transform_space * reduction ) * direction;

    const value_t beta = std::sqrt(
      ( Dimension - 1.0 ) / ( Dimension + 1.0 )
    );
    inverse_transform_space = inverse_transform_space * (
      identity + ( direction * direction.transpose() ) * ( beta - 1.0 )
    );

    const value_t reduction_step = (
      Dimension / std::sqrt( std::pow( Dimension, 2 ) - 1.0 )
    );

    reduction *= reduction_step;

    subgradient = this->calculate_subgradient( objective,
      constraints,
      optimal_point
    );

    ++iteration;

    stopping_criterion = (
      ( subgradient.norm() < accuracy ) || ( iteration >= iteration_limit )
    );
```

```
    }

    bool precondition = (
      static_cast< point_t >(
        optimal_point - initial_point
      ).norm() <= ball_radius
    );

    this->iterations = iteration;

    return precondition ? optimal_point : initial_point;
  }



  std::size_t get_iterations() const {
    return ( this->iterations );
  }



private:

  /**
   * Calculates the subgradient.
   * Choose subgradient depending on maximum of constraint functions.
   * @param objective an objective function.
   * @param constraints a list of constraint functions.
   * @param point a point to calculate subgradient.
   */
  point_t calculate_subgradient( const constraint_t& objective,
    const constraint_list_t& constraints,
    const point_t& point ) const
  {
    auto max = std::max_element( constraints.begin(), constraints.end(),
      [ & ]( const constraint_t& lhs, const constraint_t& rhs ) -> bool {
        return ( lhs.function( point ) < rhs.function( point ) );
      } );

    point_t subgradient;
    if( ( *max ).function( point ) <= value_t( 0 ) ) {
      subgradient = objective.subgradient( point );
    }
    else {
      subgradient = ( *max ).subgradient( point );
```

```
    }

    return subgradient;
  }



  std::size_t iterations;
};



#endif // ELLIPSOIDS_METHOD
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Armadillo. c++ linear algebra library . —URL: http://arma.sourceforge.net (дата обращения: 21.04.2015).
2. Boehm, B. W. Industrial software metrics top 10 list / Barry W. Boehm // IEEE Software 4. —1987. —no. 9 (September). —84–85.
3. Boost . —URL: http://www.boost.org/doc/libs/1_58_0/libs/numeric/ublas/doc/types_overview.html (дата обращения: 21.04.2015).
4. Fork–join model . — URL: http://en.wikipedia.org/wiki/Fork-join_model (дата обращения: 12.04.2015).
5. The gnu multiple precision arithmetic library . —URL: https://gmplib.org/ (дата обращения: 15.04.2015).
6. Held, M. The traveling salesman problem and minimum spanning trees. part. 2. / M. Held, R. Karp // Math. Program. —1971. —1, N 1, p. 6–25.
7. Held, M. Validation of subgradient optimization. / M. Held, P. Wolfe, H. Crowder // Math. Program. —1974. —6, N 1, p. 62–88.
8. Huang, H. Unified approach to quadratically convergent algorithms for function minimization / H. Huang. — J. Optimizat. Theory and Appl., 1970. —5, N 6.
9. It++ . — URL: http://itpp.sourceforge.net/4.3.1 (дата обращения: 21.04.2015).
10. Knuth, D. An empirical study of fortran programs / Donald Knuth // Software – Practice and Experience. —1971. —105–33.
11. Mathematical programming. Study 3. Nondifferentiable optimization / Ed. by M. L. Balinski, P. Wolfe. —Amsterdam: North–Holl and Publ. co., 1975. —178 p.
12. McCool, M. Structured Parallel Programming: Patterns for Efficient Computation / Michael McCool, James Reinders, Arch Robinson. —MK, 2013.
13. Mohrhard, M. CppUnit Documentation / Markus Mohrhard. — freedesktop.org.

14. Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools). Soc for Industrial & Applied Math / J. J. Dongarra, L. S. Duff, D. C. Sorensen, H. A. V. Vorst. — 1999.
15. Stroustrup, B. The C++ Programming Language / Bjarne Stroustrup. — Addison-Wesley, 2013.
16. Wolframalpha . — URL: http://www.wolframalpha.com (дата обращения: 10.04.2015).
17. Вальковский, В. А. Элементы параллельного программирования / В. А. Вальковский, А. Г. Марчук, Н. Н. Миренков. — М.: Радио и связь, 1983. — 240 с., интеграл.
18. Вирт, Н. Алгоритмы и структуры данных. Новая версия для Оберона + CD / Пер. с англ. Ткачев Ф. В. / Н. Вирт. — М.: ДМК Пресс, 2010. — 272 с.: ил.
19. Гершович, В. И. Метод эллипсоидов, его обобщения и приложения / В. И. Гершович, Н. З. Шор // Кибернетика. — 1982. — №5.
20. Грин, Д. Математические методы анализа алгоритмов / Д. Грин, Д. Кнут. — М.: Мир, 1987. — 120 с., ил.
21. Данилин, А. И. Основы теории оптимизации (постановки задач) [Электронный ресурс] : электрон. учеб. пособие / А. И. Данилин. — Минобрнауки России, Самар. гос. аэрокосм. ун-т им. С.П. Королева (нац. исслед. ун-т). – Электрон. текстовые и граф. дан. (1,2 МБайт). – Самара, 2011. — 1 эл. опт. диск (CD-ROM).
22. Ермольев, Ю. М. Методы стохастического программирования / Ю. М. Ермольев. — М.: Наука, 1976. — 240 с.
23. ИНТУИТ . — URL: http://www.intuit.ru/studies/courses/1156/190/lecture/4952 (дата обращения: 22.04.2015).
24. Макконнелл, С. Совершенный код. Мастер-класс / Пер. с англ. / С. Макконнелл. — М.: Издательство «Русская редакция», 2010. — 896 стр.: ил.
25. Поляк, Б. Т. Один общий метод решения экстремальных задач. – Докл. АН СССР / Б. Т. Поляк. — 1967. — 174, № 1, с. 33–36.

26. Солтер, Н. А. C++ для профессионалов.: Пер. с англ. / Николас А. Солтер, Скотт Дж. Клеппер. — М.: ООО «И.Д. Вильямс», 2006. — 912 с.: ил. – Парал. тит. англ.
27. Стецюк, П. И. Методы эллипсоидов и r-алгоритмы / П. И. Стецюк. — Нац. акад. наук Украины, Ин-т кибернетики им. В. М. Глушкова, Акад. транспорта, информатики и коммуникаций. – Кишинэу: Эврика, 2014. —488 с.
28. Хачиян, Л. Г. Полиномиальные алгоритмы в линейном программировании / Л. Г. Хачиян // Ж. вычисл. матем. и матем. физ. — 1980. — С. 51–68.
29. Шор, Н. З. Применение метода градиентного спуска для решения сетевой транспортной задачи. – В кн.: Материалы науч. семинара по теорет. и прикл. вопр. кибернетики и исслед. операций / Н. З. Шор. — Науч. совет по кибернетике АН УССР. Киев, 1962. —вып. 1, с. 9–17.
30. Шор, Н. З. О структуре алгоритмов численного решения задач оптимального планирования и проектирования: Автореф. дис. ... канд. физ.-мат. наук. / Н. З. Шор. —Киев, 1964. —10 с.
31. Шор, Н. З. Методы минимизации недифференцируемых функций и их приложения: Автореф. дис. ... докт. физ.-мат. наук. / Н. З. Шор. —Киев, 1970. —44 с.
32. Шор, Н. З. О методе минимизации почти дифференцируемых функций / Н. З. Шор // Кибернетика. —1972. —№ 4, с. 65–70.
33. Шор, Н. З. Метод отсечения с растяжением пространства для решения задач выпуклого программирования / Н. З. Шор // Кибернетика. — 1977. —№ 1, с. 94–95.
34. Шор, Н. З. Методы минимизации недифференцируемых функций и их приложения / Н. З. Шор. —Киев: Наук. думка, 1979. —200 с.
35. Юдин, Д. Б. Информационная сложность и эффективные методы решения выпуклых экстремальных задач / Д. Б. Юдин, А. С. Немировский // Экономика и мат. методы. —1976. —т. 12, вып. 2. – С. 357–369.